This electronic thesis or dissertation has been downloaded from the King's Research Portal at https://kclpure.kcl.ac.uk/portal/

KING'S College LONDON

Improved algorithmic efficiency within pattern analysis and text mining of DNA sequences

Watts, Steven

Awarding institution:
King's College London

The copyright of this thesis rests with the author and no quotation from it or information derived from it may be published without proper acknowledgement.

END USER LICENCE AGREEMENT

Unless another licence is stated on the immediately following page this work is licensed

under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International

licence. https://creativecommons.org/licenses/by-nc-nd/4.0/

You are free to copy, distribute and transmit the work

Under the following conditions:

- Attribution: You must attribute the work in the manner specified by the author (but not in any way that suggests that they endorse you or your use of the work).
- Non Commercial: You may not use this work for commercial purposes.
- No Derivative Works - You may not alter, transform, or build upon this work.

Any of these conditions can be waived if you receive permission from the author. Your fair dealings and other rights are in no way affected by the above.

Take down policy

If you believe that this document breaches copyright please contact librarypure@kcl.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 01. Mar. 2021

KING'S COLLEGE LONDON

Improved Algorithmic Efficiency Within Pattern Analysis and Text Mining of DNA Sequences

by

Steven Watts

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the

Faculty of Natural & Mathematical Sciences

Department of Informatics

June 17, 2020

Declaration of Authorship

I, Steven Watts, declare that this thesis titled, 'Improved Algorithmic Efficiency Within Pattern Analysis and Text Mining of DNA Sequences' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

Signed: Steven Watts

Date: 16th June 2020

Related Publications

This thesis represents a body of work spread across multiple research projects. As of writing, some of the results in this thesis have been published or accepted for publication. These publications are presented below, in order of the corresponding chapter within this thesis to which the publication pertains:

Chapter 3

"Client side web based application for search space reduction in approximate circular pattern matching" (Costas S. Iliopoulos, Mohammad Samir Uzzaman, M. Sohel Rahman and Steven Watts), In International Symposium on Bioinformatics Research and Applications Conference (ISBRA), 2016.

Chapter 4

"Efficient Recognition of Abelian Palindromic Factors and Associated Results" (Costas S. Iliopoulos and Steven Watts), In Artificial Intelligence Applications and Innovations (pp. 211–223), Springer International Publishing, 2018.

Chapter 5 & 6

A paper representing this work entitled "$\text{IUPAC}_{\text{PAL}}$: Efficiently Identifying Inverted Repeats in IUPAC-Encoded DNA Sequences" (Hayam Alamro, Mai Alzamel, Costas S. Iliopoulos, Solon P. Pissis and Steven Watts), has been submitted to BMC Bioinformatics.

Chapter 7

"Efficient Identification of k-Closed Strings" (Hayam Alamro, Mai Alzamel, Costas S. Iliopoulos, Solon P. Pissis, Wing-Kin Sung and Steven Watts), In Engineering Applications of Neural Networks (pp. 583–595), Springer International Publishing, 2017.

"Efficient Identification of k-Closed Strings" (Hayam Alamro, Mai Alzamel, Costas S. Iliopoulos, Solon P. Pissis, Wing-Kin Sung and Steven Watts), Accepted in IJFCS, 2019.

KING'S COLLEGE LONDON

Abstract

Faculty of Natural & Mathematical Sciences
Department of Informatics

Doctor of Philosophy

by Steven Watts

The study of data strings and its application to the processing, analysis and indexing of DNA data is an increasingly valuable area of research. This particular piece of research looks into multiple structures within data strings, and explores a related problem with the aim of reducing the time and space complexity required to obtain a solution. These problems often take the form of searching for a specific data structure, indexing a list of its occurrences or identifying the maximal cases.

Each chapter may be read as dealing with an individual algorithmic problem, though some overlap does occur within the presented problems. The focus of this research is in the algorithmic solution, and for each presented problem this is often complemented with an implementation that may be tested on real data, with the results judged by their performance.

The specific string structures that are topics of research within this thesis include: circular strings, abelian palindromes, maximal palindromes, inverted repeats, closed strings and previous factors.

Though the presented work is applicable to data strings in general, it is often the case that DNA processing provides the most direct application for such algorithmic solutions, owing to the simplicity of the alphabet and the huge scale of DNA data presently available, which lends itself well to more efficient processing methods.

Acknowledgements

Initially, I would like to express my sincere gratitude to my two advisors Prof. Costas Iliopoulos and Dr. Solon Pissis for their continued support of my research and for keeping me motivated throughout, and for their contributions in guiding me through the difficult yet exciting field of stringology research. Their guidance assisted me greatly both during the research process and up to the culmination of writing this thesis. I believe my supervisors played a significant role in guiding and mentoring me throughout my doctoral studies, and I greatly appreciate their support.

In addition to my supervisors, I would like to thank the following people for their contributions and efforts in all our joint research projects: Hayam Alamro, Mai Alzamel, Ritu Kundu, Sohel Rahman, Samir Uzzaman, Wing-Kin Sung, and Fatima Vayani. Their contributions played a vital role in furthering our research goals.

I also thank my additional fellow group mates for the great conversation and great food we shared in the past 3 years. Namely Oluwole Ajala, Lorraine Ayad, Panagiotis Charalampopoulos, Jia Gao, Chang Liu, Manal Mohamed, and Ahmad Retha. I am sure I will look back fondly on the good times we shared together and hope that more good times are yet to come.

Finally, I would like to thank my family: my parents Peter and Hildegard, my sister Nathalie and my wife Sarah, in addition to all my close friends beyond academia, for providing their support through the recent years, and making life as wonderful and enjoyable as it should be.

Contents

Chapter 1

Introduction

This thesis represents a body of work encompassing a variety of research projects undertaken within the field of stringology. As such, each chapter has been linked with a specific data structure (either pre-existing or novel), and the associated body of research presented therein.

This chapter introduces the more general background and motivation of the research undertaken as part of this thesis. The general themes and motivation are outlined in Section 1.1, with the structure of this thesis outlined in Section 1.2. Additionally, each later chapter includes a brief introduction and background for the research project undertaken which pertains to that specific chapter.

We therefore intend that the background material presented here, serves as an overview of the common ideas prevalent across all the research projects that were undertaken.

1.1 Background

Moore's Law, a product of Intel founder Gordon Moore, has been renowned for its predictive power over the past decades [1]. This law makes the assertion that the number of transistors that can fit on an integrated circuit will double every year, and along with this, an increase in computer processing power [2]. The law originally stood the test of time, though the doubling period began to quickly increase beyond the original 1 year estimate [3].

However, as with any exponential growth, the practical limitations are beginning to make themselves shown, and with this comes a gradual reduction in the rate at which computer processing power advances. Even the fundamental laws of physics themselves

will eventually impede on the law, with some estimates predicting the end of these exponential gains in the 2020s, failing some unforeseen breakthrough [4, 5].

However, this pattern of exponential growth has not slowed for one particular aspect of technological development, namely the volume of data that is being produced [6]. Within big data terminology, there are 4 particular aspects that are typically focused on: volume, variety, veracity and velocity [7].

The volume of data refers quite naturally to the numerical quantity of data, typically measured as bytes [8]. The variety refers to the increase in data formats and types of data structures for storing our data [9]. Veracity refers to security issues, and is concerned with the degree to which such large quantities of data may be trusted [10]. Finally, velocity refers to the speed at which data is transacted and exchanged, which has itself increase exponentially over recent years [11].

Within the field of bioinformatics, it is the volume of data that has increased most dramatically [12]. This has been a direct result of innovations in DNA sequencing [13]. The first human genome required more than a decade to sequence, whereas today the same feat can be accomplished in a matter of days, at a fraction of the price [14, 15]. These innovations do not necessarily correlate directly with Moore's law, and therefore we may soon find that the exponential increase in acquisition of genomic data will eventually no longer be matched by the increased ability to process that data expediently. There exists several solutions to this problem, including a paradigm by which computed results are approximated, to permit more time efficient algorithms [16].

Thus it becomes increasingly necessary to focus on optimising the algorithms with which that data is processed, rather than relying on hardware innovations that would otherwise allow researchers to develop algorithmically inefficient software. The need for algorithmic optimisation has always been present, though the motivations for this have historically varied. In the earlier days of computing, hardware limitations meant that the time and space complexity of software needed to be kept low, in order to fit within the constraints of the limited hardware. However, we now find that it is the sheer volume of data itself that might impede results, despite the rapid acceleration of both processing power and storage space.

The applications of bioinformatics to human development and quality of life are significant, and the speed at which these developments may unfold is heavily tied to the rate at which these large volumes of data may be processed and understood. These future applications include a rise in personal genomics, faster development of drugs and medicines, better understanding of disease and mutation, simulation of biological structures and numerous others [17–19].

Therefore with such goals in mind, and stemming from a grounding in theoretical computer science, this thesis covers a range of topics within the study of string searching algorithms, covering a variety of data structures. Particularly those string types which might be applied to DNA analysis, and seeks to develop algorithmic solutions with an emphasis on keeping time and space complexity low. We may find that as the increase of technical limitations of computers begins to decelerate, the need for optimisation becomes ever more valuable, less of a theoretical pursuit, and with real world practical gains to be made within the study of bioinformatics.

1.2 Thesis Structure

We have structured this thesis, such that each chapter is devoted to a self-contained problem within stringology research, with a small amount of overlap between chapters. Thus the structure represents the methodology that was undertaken during the research period, namely working on various interconnected problems as opposed to a single unifying problem, though still within the scope of a single string searching theme. The chronology of chapters has been chosen such that concepts relating to multiple chapters are introduced steadily, with those chapters requiring a larger body of prior knowledge appearing later in the thesis. With the exception of the final Chapter 8, each represents a completed body of work, with a clearly stated research goal and conclusion.

Each chapter presents an overview of an individual research goal, a background into the problem, an introduction to any necessary terminology, followed by additional theoretical tools. We then formally state the algorithmic problem, followed by either a presentation of an efficient algorithmic solution or a practical implementation. Where relevant, this is concluded with an analysis of the algorithmic efficiency and any further concluding remarks.

The title of each chapter has been chosen to reflect a specific structure or concept to which the algebraic problem relates. In this way, a clear distinction is made as to the application of each individual chapter. However it should be made clear that the ultimate application of each chapter often pertains to DNA data processing, thus all presented examples and diagrams are presented with this in mind, and thus the title of this thesis: "Improved Algorithmic Efficiency Within Pattern Analysis and Text Mining of DNA Sequences". The contents of each chapter are summarised below:

Chapter 3: Circular Strings

We present an implementation of a new web tool used to solve a string searching problem described as Approximate Circular Pattern Matching (ACPM). A web tool is presented which is capable of running the proposed algorithm on a client machine, reducing the need for data transfer when handling large quantities of data. We analyse the speed of the web tool, for the purposes of comparison to other methods.

The algorithms used make use of a filtering process to reduce the initial search space. The result is a tool that successfully solves the ACPM problem with low algorithmic complexity, and removes the necessity for significant data transfers over a network. The source code of the implementation is made publicly available.

Chapter 4: Abelian Palindromes

We define a new data structure, namely the *abelian palindrome*. The combinatorial problem of identifying abelian palindromes as factors is then solved, through the use of an algorithm which makes use of an additional novel data structure, the *prefix parity integer array*.

We create a further algorithm to generate a data structure defined as the *abelian palindromic array*, which summarises the properties of a string as they relate to abelian palindromic factors. The pseudocode is presented for both algorithms.

Chapter 5: Maximal Palindromes

We describe research working on degenerate strings, and the creation of an algorithm that is capable of producing a sequence of maximal palindromic factors corresponding to a given degenerate string. In addition, we describe an algorithm generating a list of all maximal palindromes, along with the pseudocode of an implementation. The algorithms assume a known or approximated limit of non-solid symbols within a given degenerate string.

We additionally describe an algorithm, which has the ability to determine a maximal palindromic factorisation of a degenerate string. These algorithms are implemented, and their efficacy explored through a series of tests. This chapter serves as a precursor to Chapter 6, which goes into greater depth on the study of palindromes in degenerate strings and provides further applications.

Chapter 6: Inverted Repeats

We present a command-line software implementation capable of identifying inverted repeats in DNA sequences. The implementation assumes the use of IUPAC encoded characters, and allows for possible gaps and mismatches in characters.

We perform an analysis, demonstrating that our implementation IUPACpal performs favourably when compared to a popular alternative EMBOSS [20, 21]. The software is additionally shown to have the capability of identifying previously unidentified inverted repeats when compared with EMBOSS, and performs this task with a far reduced run-time.

We detail the data pipeline and workflow, and the source code is made publicly available for use.

Chapter 7: Closed Strings

We address an extension of the problem of identifying closed strings, by introducing the concept of k-closed strings, in which a specified level of approximation is permitted, in the form of mismatching characters.

We present an algorithm, which identifies whether or not a given string is k-closed and additionally specifies the associated border. This algorithm makes use of newly introduced data structures. We present the pseudocode of the algorithm which achieves this, along with some proof-of-concept experimental results.

Chapter 8: Previous Factors

We present initial findings of an attempt to create an algorithm that may generate the *Longest Previous Factor* array for degenerate strings efficiently. As a work in progress, the format of this chapter deviates from previous chapters.

We present initial attempts to improve algorithmic efficiency beyond the trivial solution, along with an outline of suggested improvements, which provide potential starting points for further research to be undertaken.

Chapter 2

Background Theory

The field of stringology is a scientific domain that sits neatly at the intersection between mathematics and computer science. There is a great accumulation of work within this field, with various algorithmic solutions to problems that all ultimately relate to the processing or analysis of text [22–24].

With this great body of preexisting work, comes a set of common conventions and terminology which must first be clarified before proceeding to the novel research presented in later chapters. This chapter covers the most common string related structures, including a firm definition of the data string itself. Since the goal of each later chapter is the reduction of space and time complexity, there is additional clarification provided to the standard definitions which clarify how this complexity is measured and compared.

2.1 Algorithmic Complexity

Throughout the various problems addressed in this thesis, we will frequently encounter the concepts of time and space complexity. These two related concepts, allow us to consider the physical limitation of computational power, when judging the efficiency of any theoretical algorithm.

Space complexity relates to the physical hardware of the computational machine, and considers the physical storage space required to store any data structures or intermediate calculations during the execution of an algorithm. Time complexity relates to the running time of a machine when executing an algorithm and is additionally dependant on the computational power of the machine.

When we consider space complexity and time complexity, there are 4 popular notations used to describe bounds of algorithmic complexity, in terms of the size of input data

[25–27]. By popular convention we use n to refer to this input size. We present these various notations in Def. 2.1.

Definition 2.1. Given a function $T(n)$, which describes the space or time consumption of an algorithm as a function of input size n, we can characterise T as being a member of one of the following sets defined in terms of a function $f(n)$:

$$T(n) \in \mathcal{O}(f(n)) \iff \exists\, c, n_0 : T(n) \leq cf(n) \;\; \forall n \geq n_0 \tag{2.1a}$$

$$T(n) \in \Omega(f(n)) \iff \exists\, c, n_0 : T(n) \geq cf(n) \;\; \forall n \geq n_0 \tag{2.1b}$$

$$T(n) \in \Theta(f(n)) \iff T(n) \in \mathcal{O}(f(n)) \wedge T(n) \in \Omega(f(n)) \tag{2.1c}$$

$$T(n) \in o(f(n)) \iff T(n) \in \mathcal{O}(f(n)) \wedge T(n) \notin \Theta(f(n)) \tag{2.1d}$$

All 4 notations in Def. 2.1 relate to the asymptotic behaviour of $T(n)$, describing behaviour that occurs for large values of n. This corresponds to the practical application of algorithms, in which performance on increasingly larger quantities of data must be considered. The intuition behind each notation in Def. 2.1 may be expressed as follows:

$T(n) \in \mathcal{O}(f(n)) : f(n)$ describes the upper bound of $T(n)$

$T(n) \in \Omega(f(n)) : f(n)$ describes the lower bound of $T(n)$

$T(n) \in \Theta(f(n)) : f(n)$ describes the exact bound of $T(n)$

$T(n) \in o(f(n)) : f(n)$ describes the upper bound of $T(n)$ but never equals $T(n)$

Within this thesis, we will focus solely on the most commonly used notation, namely Def. 2.1a, dubbed "big O" notation. This derives from a desire to formally define the "worst-case scenario" for an algorithmic solution, and provides a practical guideline to the efficiency of an algorithmic solution. Although $f(n)$ as used in Def. 2.1 may be defined as any real function, we typically use one of several commonly used functions, which we present in Fig. 2.2.

Depending on the complexity of an algorithmic problem, we set our sights on a solution which corresponds to a slowly growing function $f(n)$ accordingly. The seldom encountered best case is an $\mathcal{O}(1)$ complexity, though for any solution to a significant problem this will rarely be encountered and often requires a degree of approximation [28]. More realistically, for any algorithmic problem expected to operate on large data sets, a complexity of $\mathcal{O}(n)$ or less is a significant achievement. However some problems may require $\mathcal{O}(n \log(n))$ space or time, and yet others more complexity still.

In some cases, it may be difficult to determine the complexity of a particular solution. Sometimes the number of resources used by each individual operation may vary [29, 30], and the complexity of the whole must be determined through an amortised analysis,

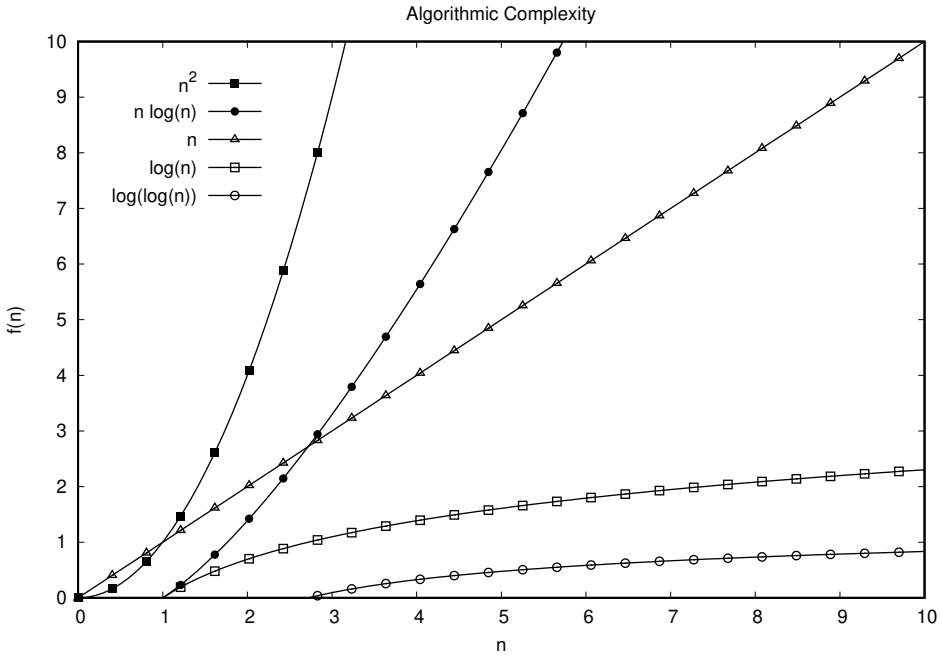

FIGURE 2.2: Comparison of commonly used complexity functions.

rather than a simple summing of resource usage of individual operations. Complexity of a solution may be determined by experimenting with an real-world implementation, particularly for solutions with some practical application. However, we strive to provide a theoretical complexity to all solutions presented in this thesis. Only a theoretical analysis can provide a definitive mathematical proof of the efficiency of a solution. It is important to note however, that implementations may differ from theory, particularly when the hidden constant c in Def. 2.1a is large [31].

2.2 Strings

We begin with basic definitions and notation from [32]. Let $x = x[0]\,x[1]\,\ldots\,x[n-1]$ be a *string* of length $|x| = n$ over some alphabet Σ. We use $x[i]$ to refer to the ith character of x using 0-based indexing. Two strings x and y are considered equal if they are of the same length and represent the same sequence of characters:

$$x = y \iff |x| = |y| \wedge x[i] = y[i] \quad \forall i \in [0, |x| - 1]$$

For two positions i and j of x, we denote by $x[i\,..\,j] = x[i]\,x[i+1]\ldots x[j-1]\,x[j]$ the *factor* (sometimes called *substring*) of x that starts at position i and ends at position j inclusively. The length of such a factor $|x[i\,..\,j]|$ is therefore of length $j - i + 1$, and we use the symbol ε to refer to an *empty string* of length 0.

The *Hamming distance* between two strings x and y both of length n, is the number of positions i such that $x[i] \neq y[i]$ where $0 \leq i \leq n-1$ [33]. Given an integer $k \geq 0$, we write $x \equiv_k y$ or equivalently say that x k-matches y, if the Hamming distance between x and y is at most k. In biology, the Hamming distance is popularly referred to as the *mutation distance*.

We define a *prefix* of a string x as any factor that starts at position 0, i.e. $x[0\,..\,j]$ for some j. We define a *suffix* of a string x as a factor that ends at position $n-1$, i.e. $x[i\,..\,n-1]$ for some i. We use the term *proper prefix* and *proper suffix* to refer to a prefix and suffix respectively that are not equal to the full string x, i.e. with a size smaller than $|x| = n$. An example of a prefix and suffix is shown in Fig. 2.3.

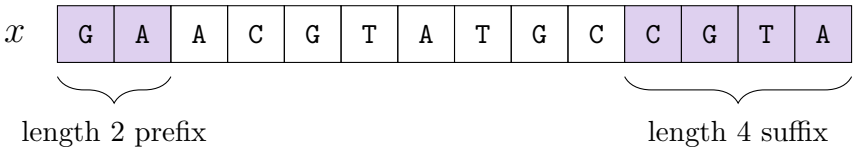

FIGURE 2.3: Example of prefix and suffix.

Let y be a string of length m with $0 < m \leq n$. We say that there exists an *occurrence* of y in x, or more simply, that y *occurs in* x, when y is a factor of x. Every occurrence of y can be characterised by a starting position in x. Thus we say that y occurs at the *position* i in x when $y = x[i\,..\,i+m-1]$. An example of string occurrence is shown in Fig. 2.4.

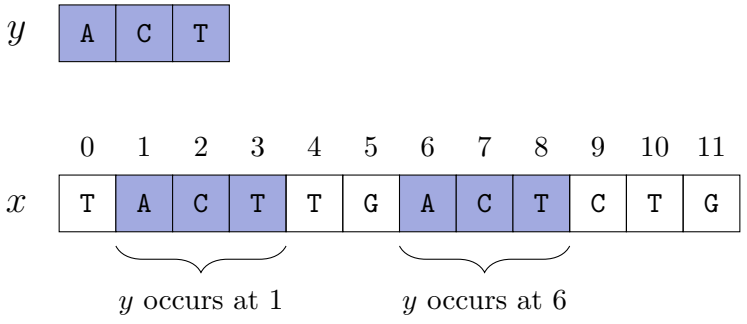

FIGURE 2.4: Example of string occurrence.

Occasionally we may find it useful to substitute characters of an alphabet with numerical values. We call an alphabet Σ an *ordered alphabet* when the characters of the alphabet may by placed in a unique lexicographical ordering [34, 35]. Additionally, we call a finite ordered alphabet Σ of size n an *integer alphabet* when every character may be replaced

by its lexicographical 0-based rank in such a way that the resulting alphabet consists of the integers in the range $\{0, \ldots, n-1\}$.

For an integer alphabet, we use $\textsc{ord}(\sigma)$ to refer to the lexicographical rank of the character σ. We use $\Sigma[i]$ to refer to the ith character of Σ, i.e. $\Sigma[\textsc{ord}(\sigma)] = \sigma$. For example, given the alphabet $\Sigma = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$ we have $\textsc{ord}(\texttt{G}) = 2$ and $\Sigma[2] = \texttt{G}$.

Finally, we define the function *longest common prefix* of two strings x and y, denoted $\text{lcp}(x, y)$ [36]. This allows us to compare the commonality of the prefixes of two strings or potentially two substrings of the same string. Specifically, the function returns the longest prefix of x which also appears as a prefix of y, formally defined in Def. 2.5.

Definition 2.5. For two given strings x and y, we define the longest common prefix $\text{lcp}(x, y)$ as follows:

$$\text{lcp}(x, y) = \max(\{l : x[0\,..\,l-1] = y[0\,..\,l-1]\} \cup \{0\})$$

Note that when no common prefix occurs between two strings, the lcp function returns a value of 0. An example of a longest common prefix is shown in Fig. 2.6.

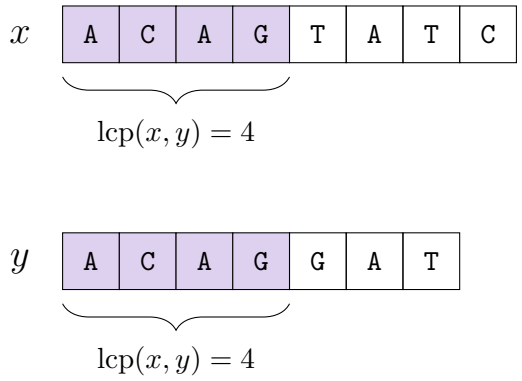

FIGURE 2.6: Example of the longest common prefix of two strings x and y.

A closely related function is the *longest common extension*, which we will denote lce [37]. This takes two prefixes of the same string and finds their longest common prefix. Whereas the lcp function accepts two strings as arguments, the lce function accepts a single string, and two index locations within the string representing two prefixes. We define the lce function formally in Def. 2.7.

Definition 2.7. For a given string x of length n and two indexes, we define the longest common extension $\text{lce}(x, i, j)$ as follows:

$$\text{lce}(x, i, j) = \max(\{l : x[i \,..\, i + l - 1] = x[j \,..\, j + l - 1]\} \cup \{0\})$$

The function in Def. 2.7 provides the length of the longest common prefix of two specified suffixes of a string x located at positions i and j within the string x. Note that when there is no common prefix of any length between the ith and jth suffix of a string, the lce function returns a value of 0. An example of a longest common extension is shown in Fig. 2.8.

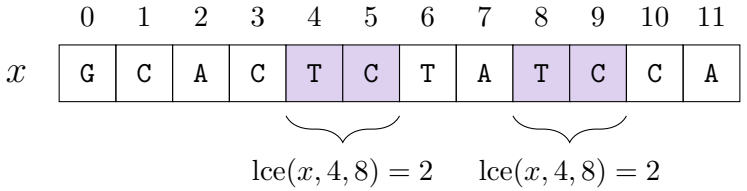

FIGURE 2.8: Example of the longest common extension of a string x at $i = 4, j = 8$.

The lce function may be generalised, by allowing up to a fixed number of mismatches when considering the extension, which we will denote by the function lce_k where k is the maximum number of mismatches. The lce_k function accepts a single string, and two index locations within the string representing two prefixes and an integer k. We define the lce_k function formally in Def. 2.9.

Definition 2.9. For a given string x of length n and two indexes, and an integer k, we define the longest common extension with k mismatches $\text{lce}_k(x, i, j)$ as follows:

$$\text{lce}_k(x, i, j) = \max(\{l : x[i \,..\, i + l - 1] \equiv_k x[j \,..\, j + l - 1]\} \cup \{0\})$$

Note that when k mismatches are permitted, the minimum length of the longest common extension is k. An example of a longest common extension using mismatches is shown in Fig. 2.10.

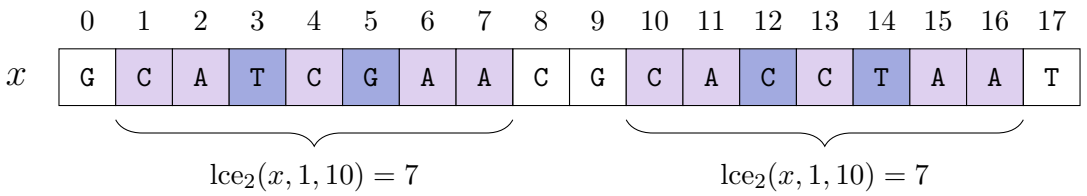

FIGURE 2.10: Example of the longest common extension of a string x at $i = 1, j = 10$ with permitted mismatches $k = 2$.

In later chapters, Def. 2.7 may be extended further to consider alternate matching conditions and approximations. In particular, Chapter 5 will make use of an additional parameter known as a *matching table* to further generalise the conditions by which two characters match, when considering the longest common extension.

2.3 Suffix Trees

The most important data structure that will be encountered throughout this thesis is the *suffix tree* [38]. A suffix tree is constructed with respect to a specific string x, and allows for efficient searching of substrings within the string x. As a direct result, there are several other operations that may be performed efficiently on the string x, once its suffix tree is constructed.

To understand the structure of a suffix tree, we first define a related structure, namely the *trie* [39]. A trie is a tree data structure, which stores a set of string keys over some alphabet Σ, such that each leaf in the tree corresponds to a key. Every edge in the tree is labelled with some character from the alphabet Σ, and every path from root to leaf within the tree corresponds to a key. The key corresponding to a path, is determined by concatenating the characters associated with the edges in the path, in the order in which they are encountered. For a given internal node, within the set of connecting edges to lower tree levels, there may never be multiple instances of the same character labelling an edge, i.e. a given string may never correspond to more than one valid path in the trie. An example of a trie is shown in Fig. 2.11.

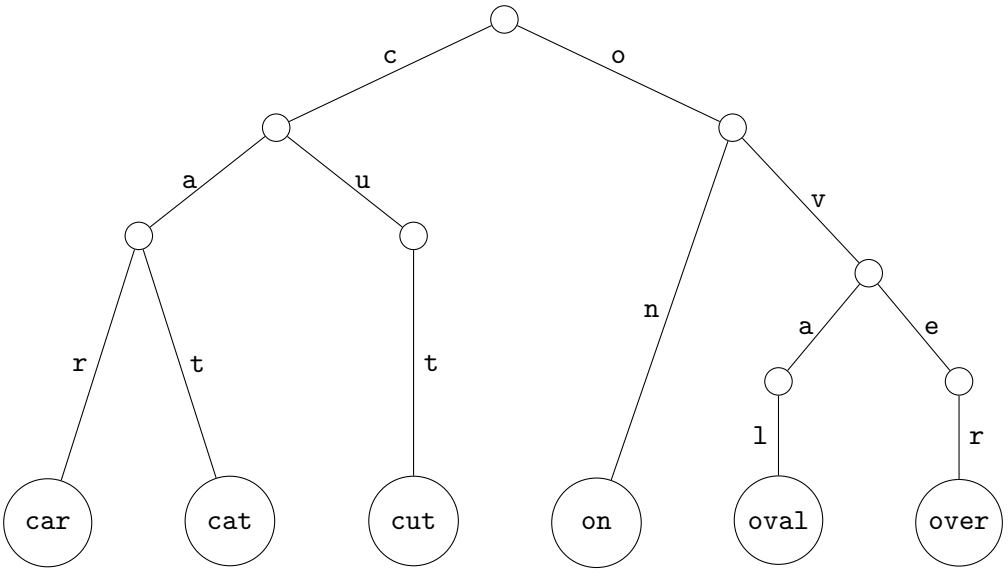

FIGURE 2.11: Example of a trie for the set of keys $\{\texttt{car}, \texttt{cat}, \texttt{cut}, \texttt{on}, \texttt{oval}, \texttt{over}\}$ over the alphabet $\Sigma = \{\texttt{a}, \texttt{c}, \texttt{e}, \texttt{l}, \texttt{n}, \texttt{o}, \texttt{r}, \texttt{t}, \texttt{u}, \texttt{v}\}$.

Stated succinctly, a suffix tree of a string x is the trie of the set of all suffixes of x, with some additional implementation details that improve storage efficiency and prevent duplicate keys.

For example, consider the string $x = \texttt{CAACAACC}$ of length 8. In this case, a suffix tree on x is a trie over the keys $\{\texttt{C}, \texttt{CC}, \texttt{ACC}, \texttt{AACC}, \texttt{CAACC}, \texttt{ACAACC}, \texttt{AACAACC}, \texttt{CAACAACC}\}$, the set of all non-zero length suffixes of x.

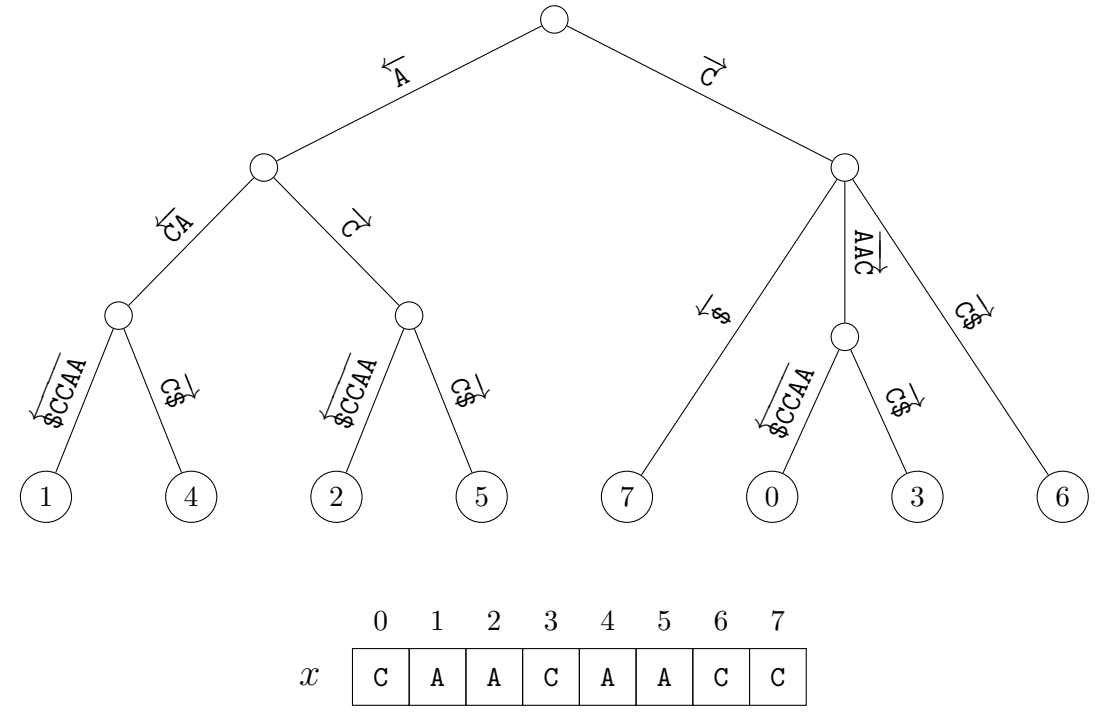

FIGURE 2.12: Example of a suffix tree for the string $x =$ `CAACAACC`.

Some additional implementation details are worthy of note; The tree edges in a suffix tree are compressed, such that when a series of connected edges in the original trie define a single path, the edges are compressed into a single edge, with a label that corresponds to the concatenation of the original edge labels. For instance in Fig. 2.12, we see an edge label `CA`, which results from the compression of two sequential edges originally labelled `A` and `C`.

To ensure every suffix of x corresponds to a unique path and a unique leaf node, the string x is appended with the character `$` before using the suffixes to construct the suffix tree. This character `$` is defined to not occur within the alphabet Σ over which x is defined, and matches no characters in Σ other than itself. In this way, every suffix is terminated with the character `$`, and in conjunction with the fact that no two suffixes are of the same length, this guarantees a unique leaf node to precisely one suffix of x.

Additionally, the implementation of the edge labels may be optimised, by using the original string x as a reference point to determine edge labels rather than storing the full text of each label [40, 41]. This is done by using an appropriate pair of positive integers (i, l) to reference any required factor of x, where i is the starting index of the factor in x and l is the length of the factor. Therefore a pair (i, l) refers to the factor $x[i \dots i + l - 1]$. For instance in Fig. 2.12, we see an edge label `AAC`, which an efficient

implementation may represent as the pair of indexes $(1, 3)$, referring to the substring $x[1\,..\,3]$ starting at index 1 and with length 3. By storing the string x of length n alongside the suffix tree, we may represent all edge labels using a pair of integers from the set $\{0, \ldots, n\}$. Note that within this efficient storage model, the otherwise undefined index position n of the string x, is defined to hold the character \$.

The details provided thus far present only a high-level overview, however further details on how a suffix tree may be efficiently implemented are well documented [42]. The key result of note, is that a single instance of a suffix tree over a string x of length n, may be constructed in $\mathcal{O}(n)$ time and requires $\mathcal{O}(n)$ storage space. Once a suffix tree of a string x is constructed, we may perform various operations on the tree, which may answer queries about the nature of x. We detail examples of such queries in Table 2.1 [43].

The suffix tree is a fundamental data structure within the field of stringology, and the use of this data structure will be a recurring theme within the algorithms presented in this thesis.

Operation	Preprocessing		Query
	Time	Space	Time
Check if a substring of size m occurs within a string x of size n.	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(m)$
Determine up to z occurrences of a substring of size m within a string x of size n.	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(m+z)$
Determine the longest repeated substring within a string x of size n.	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Build the suffix array (see Section 2.4) of a string x of size n.	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Determine the longest common substring of two strings x, y of size n, m respectively.	$\mathcal{O}(n+m)$	$\mathcal{O}(n+m)$	$\mathcal{O}(n+m)$

TABLE 2.1: Common string operations making use of suffix trees in preprocessing, with associated time and space complexity.

2.4 Suffix Arrays and LCP Arrays

Having discussed suffix trees, we now consider *suffix arrays* (SA) and *longest common prefix* (LCP) *arrays*. These arrays are a pair of related data structures that in addition to other applications, may be used as an alternative to suffix trees [44].

Given a string x of length n, the associated suffix array SA_x is an array of length n, where each entry $\mathrm{SA}_x[i]$ contains the starting index of the ith smallest suffix in x, where we define the ordering from smallest to largest of the suffixes in terms of their lexicographical order. Thus the suffix array SA_x represents a permutation of the set $\{0, \ldots, n-1\}$. An example of such a lexicographical ordering is shown in Table 2.2. We formally define the suffix array in Def. 2.13.

Definition 2.13. Given a string x of length n, we define the suffix array SA_x of x as the unique integer array permutation of the set $\{0, \ldots, n-1\}$ such that:

$$x[\mathrm{SA}_x[i-1]\,..\,n-1] < x[\mathrm{SA}_x[i]\,..\,n-1] \quad \forall i \in \{1, \ldots, n-1\}$$

We additionally define the *inverse suffix array* (iSA) as the inverse of the suffix array SA, i.e. $\mathrm{iSA}_x[i] = j \iff \mathrm{SA}_x[j] = i$.

i	$\mathrm{SA}_x[i]$	$x[\mathrm{SA}_x[i]\,..\,n-1]$
0	4	AATCGATGATC
1	0	AATGAATCGATGATC
2	12	ATC
3	5	ATCGATGATC
4	1	ATGAATCGATGATC
5	9	ATGATC
6	14	C
7	7	CGATGATC
8	3	GAATCGATGATC
9	11	GATC
10	8	GATGATC
11	13	TC
12	6	TCGATGATC
13	2	TGAATCGATGATC
14	10	TGATC

TABLE 2.2: Example of SA_x for the string $x =$ AATGAATCGATGATC.

Because no two suffixes of the same string can have the same length, no two suffixes of the same string can be equal. Thus Def. 2.13 correctly asserts that the suffix array of a string is unique, corresponding to a unique ordering of suffixes.

We now define the LCP array which is defined with reference to the SA array. The LCP array of a string stores the longest common prefix between each pair of consecutive suffixes of the suffix array. We state this formally in Def. 2.14.

Definition 2.14. Given a string x of length n, we define LCP_x of x accordingly:

$$\text{LCP}_x[0] = 0$$
$$\text{LCP}_x[i] = \text{lcp}(x[\text{SA}_x[i-1]\,..\,n-1], x[\text{SA}_x[i]\,..\,n-1]) \quad 0 < i < n$$

	0	1	2	3	4	5	6	7
x	G	A	T	A	G	A	C	A
SA_x	7	5	3	1	6	4	0	2
LCP_x	0	1	1	1	0	0	2	0

FIGURE 2.15: Example of SA_x and LCP_x for the string $x =$ `GATAGACA`.

2.5 Range Minimum Queries

A data structure which is frequently used in conjunction with suffix trees is the RMQ structure used to answer *range minimum queries* (RMQs) [45]. This is a data structure derived from an array of integer values A, that is able to determine the index of the minimal value in A that occurs between any two given indexes i and j within A. This structure may be used for several applications, including the determination of the lowest common ancestor of two leaf nodes in a suffix tree. We formally define the behaviour of the RMQ data structure in Def. 2.16.

Definition 2.16. Given an array of integers A of length n, RMQ_A is any data structure of size $\mathcal{O}(n)$ which may answer in $\mathcal{O}(1)$ time, any query of the form:

$$\text{RMQ}_A(i,j) = \min\{k : A[k] = \min\{A[i], \ldots, A[j]\}, i \leq k \leq j\} \quad 0 \leq i \leq j \leq n-1$$

Here we have defined RMQ in terms of its application and performance rather than its specific implementation. There exist several implementations of data structures that answer RMQs with various space and time complexities. Within Def. 2.16 we refer to a data structure defined to have constant time queries and linear space consumption in terms of the size of the original array A. Because RMQs are so applicable and well studied, there exist such efficient implementations which primarily make use of Cartesian trees [46].

For example, if we create an RMQ structure over the lcp array of a string x, we may use this to efficiently determine the longest common extension $\text{lce}(x, i, j)$ for any pair of suffixes of x at positions i and j. We provide an example of a range minimum query as applied to an array of integers in Fig. 2.17.

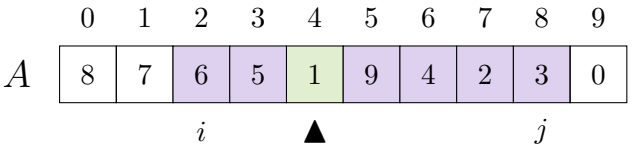

FIGURE 2.17: Example of a range minimum query on A for $i = 2$, $j = 8$. The minimal value within the range is found at index 4 with value 1.

Note that in the scenario where more than one minimal value exists within the bounded range, there are varying conventions on how to handle the tie-break [47]. We will use the convention that the leftmost index satisfying the condition of having a minimal value will be used, when multiple satisfying indexes exist.

Chapter 3

Circular Strings

We now focus on the first problem of interest within this thesis, specifically concerning circular strings and their application. Details will be provided concerning the implementation and performance of a new web tool used to determine solutions to Approximate Circular Pattern Matching (ACPM) problems, which directly make use of circular strings.

The circular pattern matching problem consists of finding all occurrences of all rotations of a given pattern p of length m within a given text t of length n [48]. Exact matches will be those that match on all locations, whereas approximate matches are those that match on all but k positions, where k is a specified approximation parameter.

The algorithms used are based on a variation of previously developed methods [49, 50] and are implemented as a JavaScript based client-side web tool to facilitate the handling of big data sets. When compared to methods that rely on transmission of big data over a network, the web tool shows a significant improvement.

3.1 Background

Simple pattern matching is a fundamental problem in computer science with various applications in branches of engineering, informatics, genetics and biology [51]. The basic statement of the problem considers a pattern p of length m and a pattern t of length n. Both pattern and text are comprised of characters taken from a mutual alphabet of finite characters Σ, for example the DNA alphabet.

The goal of pattern matching is to determine the locations within the text t at which the pattern p occurs as a substring. A location corresponds to the index of t at which an

occurrence of p begins. Most practical approaches to this problem demand an additional element of approximation, in order to handle various types of noise or other errors that may occur within the data [52]. The basic pattern matching problem has been well studied, with many variants to the problem being developed to deal with different models of approximation [53]. We will make use of the Hamming distance method of approximation.

In addition to allowing for approximate matches, the problem can be further altered to treat search patterns as circular strings, to give us the Approximate Circular Pattern Matching problem. In this version of the problem, the pattern P is a circular string. Such circular strings can be visualised as a series of characters which loop back around on themselves, such that there is no clearly defined start or end of the string. The formal definition of circular strings is given in Def. 3.1.

With this definition, we can address the problem of locating circular patterns in a text. This version of the problem requires an algorithm that reports all locations of all rotations of pattern p in a text t. Note that extending the problem to allow for approximate matches results in a more complex algorithm to determine match locations, but will result in a more practically useful solution.

Finding circular patterns is a difficult combinatorial problem that requires a well developed algorithm to solve efficiently. The problem is interesting from a theoretical standpoint, and relevant to work in mathematics, geometry and biological computation [54–56]. There is an additional practical consequence, aided by the fact that circular patterns are one of many patterns that may occur in the DNA of various genetic structures. These include viruses [57, 58], bacteria [59], cells of eukaryotes [60] and archaea [61]. Organisms with such circular structures may be more readily identified and studied through algorithms catering specifically to circular string problems. Circular strings have also been considered when developing algorithms for both pairwise and multiple sequence alignment. Algorithms have been presented which apply these techniques specifically to circular strings [62, 63]. There has additionally been work to create efficient algorithms for determining optimal alignments and consensus sequences for multiple circular sequences [64].

Consider for example a biologist who wishes to determine whether or not a circular genetic pattern associated with a virus occurs within an individual genome. Additionally the biologist wishes to know the locations within the genome at which this virus appears. A circular string can represent an arbitrary rotation of the virus pattern and the biologist wishes to locate any possible rotations of this pattern in the genome text. However in this practical case, the biologist is aware that both the pattern and text data may exhibit errors. This may be due to erroneous sampling methods, incomplete

data, sensor noise and various other possible defects [65]. Therefore it is necessary to consider Approximate Circular Pattern Matching (ACPM) which allows for practical searching methods that are capable of ignoring a specified threshold of errors caused by the limitations of accuracy in data collection. For this reason we will focus on the ACPM problem.

3.2 Problem Outline

3.2.1 Terminology

Small mutations and errors should be expected when searching for occurrences of a particular circular virus in a DNA sequence. The Hamming distance is a commonly used measure of such mutations. If we set $k = 0$ then we need only use an algorithm which reports exact matches. Alternatively if $k > 0$ we must use an ACPM algorithm to return a larger number of reported locations, in which the reported matches may mismatch the pattern at up to k positions. This would prevent the failed reporting of a virus present in a DNA sequence due to small discrepancies.

Given a fixed pattern p of length m, we may form a circular pattern denoted $C(p)$. Within the field of stringology, this may also be referred to as a *conjugacy class*. This can be considered as a set of m strings corresponding to the set of m possible rotations of p. A rotation of p is a shift of the characters of p to the left with characters wrapping around. We use p^r where $r \in \{0, \ldots, m-1\}$ to denote the rth rotation in which characters are shifted r places to the left. It follows from the definition that $p^0 = p$ and there are m possible rotations for a pattern p of length m. We formally define this in Def. 3.1.

Definition 3.1. Given a string x of length n, we use x^i to refer to the string of length m known as the ith rotation of x, where $x^i = x^{i \mod m}$ as defined below:

$$
\begin{aligned}
x^0 &= x[0]\ x[1]\ x[2]\ \ldots\ x[n-3]\ x[n-2]\ x[n-1] \\
x^1 &= x[1]\ x[2]\ x[3]\ \ldots\ x[n-2]\ x[n-1]\ x[0] \\
x^2 &= x[2]\ x[3]\ x[4]\ \ldots\ x[n-1]\ x[0]\ x[1] \\
&\vdots \\
x^{n-1} &= x[n-1]\ x[0]\ x[1]\ \ldots\ x[n-4]\ x[n-3]\ x[n-2]
\end{aligned}
$$

Consider the simple genetic pattern $p =$ `AGTTAGCA`. In this case the length of p is $m = 8$, therefore p will have 8 unique rotations. The corresponding circular pattern of p can be represented as the set $C(p)$. Each member of this set is a rotation of p. An example of the rotations for this particular string is shown in Fig. 3.2 and 3.3.

p^0	A	G	T	T	A	G	C	A
p^1	G	T	T	A	G	C	A	A
p^2	T	T	A	G	C	A	A	G
p^3	T	A	G	C	A	A	G	T
p^4	A	G	C	A	A	G	T	T
p^5	G	C	A	A	G	T	T	A
p^6	C	A	A	G	T	T	A	G
p^7	A	A	G	T	T	A	G	C

FIGURE 3.2: Rotations of the string $p =$ `AGTTAGCA`.

FIGURE 3.3: Graphical representation of the circular string in Fig. 3.2.

Because the primary applications of this work are within the field of genetics, we focus on developing a tool that operates over the DNA alphabet. Specifically the alphabet $\Sigma = \{$`A`, `C`, `G`, `T`$\}$. In this approach, each character of the alphabet is associated with a numeric value.

Characters are assigned a unique number from the range $[1, |\Sigma|]$. This numbering may be arbitrary, simply assigning a unique integer to each of the characters in Σ in accordance

with their ranking in lexicographical order. We use num(σ) where $\sigma \in \Sigma$ to denote the numeric value of the character σ. Therefore for the DNA alphabet we have the following numeric associations:

$$\text{num}(\texttt{A}) = 1 \quad \text{num}(\texttt{C}) = 2 \quad \text{num}(\texttt{G}) = 3 \quad \text{num}(\texttt{T}) = 4$$

For a string x, we use the notation x_N to denote the numeric representation of the string x and $x_N[i]$ denotes the numeric value of the character $x[i]$. For example if $x[i] = \texttt{G}$ then $x_N[i] = \text{num}(\texttt{G}) = 3$. The definition of circular strings and their rotations also applies naturally to their numeric representations.

Suppose we have a pattern $p = \texttt{ATCGATG}$. In this case the numeric representation of p is $p_N = 1423143$. And this numeric representation has the rotations as shown in Fig 3.4.

p^0	1	4	2	3	1	4	3
p^1	4	2	3	1	4	3	1
p^2	2	3	1	4	3	1	4
p^3	3	1	4	3	1	4	2
p^4	1	4	3	1	4	2	3
p^5	4	3	1	4	2	3	1
p^6	3	1	4	2	3	1	4

FIGURE 3.4: Numeric rotations of the string $p = \texttt{ATCGATG}$.

It will prove useful to keep in mind the maximum numerical difference over our alphabet. This represents the maximum possible difference in value between any two characters, and is defined within this chapter as M such that:

$$M = max\{\text{num}(a) - \text{num}(b) : a, b \in \Sigma\}$$

In the case of our numbering for the DNA alphabet, $M = |4 - 1| = 3$.

3.2.2 Problem Statement

We now formally define the problem addressed within this chapter, namely Approximate Circular Pattern Matching (ACPM):

> Approximate Circular Pattern Matching
>
> **Input:**
> A pattern p of length m, a text t of length $n > m$, and an integer error threshold $k < m$.
>
> **Output:**
> The set of index locations within t such that some rotation of the pattern p occurs within Hamming distance k of the substring at that location. This corresponds to the set $\{0 \leq i < n - m + 1 : \exists r \text{ s.t. } p^r \equiv_k t[i \,..\, i + m - 1]\}$.

Note that the full problem as stated will be solved, however the focus will be on providing an efficient practical implementation that may reduce the initial search space required to determine the correct output to the above problem.

3.3 Solution

3.3.1 Tools

The proposed implementation initially makes use of filtering techniques, to reduce the search space. A filter represents a function mapping from a string to some value or set of values. There are 2 properties which must be satisfied by a filter function F for it to have practical use in reducing the search space quickly:

Invariance:

The filter F maps to the same value for all rotations of a string. Formally we say that a function F is invariant for strings of length m if $F(x) = F(x^i)$ for all $0 \leq i < m$ for all possible strings x of length m. A function F is invariant in general if it is invariant for strings of length m for all $m \geq 0$.

$\mathcal{O}(1)$ **Rolling Calculation:**

We construct the filter F in such a way that it can calculate the mapping of a text window in constant time based on the mapping of the previous text window.

> Formally we say that a function F is capable of $\mathcal{O}(1)$ rolling calculation if given any m-sized text window $t[i\,..\,i+m-1]$ and the value of its mapping $F(t[i\,..\,i+m-1])$, it is possible to calculate the mapping of the subsequent text window $F(t[i+1\,..\,i+m])$ in $\mathcal{O}(1)$ time, and this holds for any given string t.

Given a filter F that satisfies these properties, it can be used to remove possible windows of the text from being considered as matches. This is because the mapping of a matching text window must necessarily be equal to the mapping of the pattern. If this condition is violated, we may definitively rule out that window. This makes it possible to rule out many text windows in a total of $\mathcal{O}(n)$ time for a text t of length n.

In the case of approximate matching, the condition that the mapping of p and a text window t must be equal, may be relaxed appropriately. The approximation threshold k will determine how this condition is relaxed. There are 3 specific filters that have been identified and implemented.

Definition 3.5 (Value Sum Filter)**.** Using the num notation as defined in Section 3.2.1, we can define the value sum filter F_1 on a string x of length m as:

$$F_1(x) = \sum_{i=0}^{m-1} x_N[i] \quad F_1 : \Sigma^* \mapsto \mathbb{N}$$

This filter takes the numerical summation of the string once characters have been converted into their corresponding numerical values. With respect to the problem defined in Section 3.2.2, we can rule out a match with a window starting at position i of the text t where the following does not hold:

$$|F_1(t[i\,..\,i+m-1]) - F_1(x)| \leq kM$$

The constant M represents the maximum possible absolute change in the value of $F_1(x)$ induced by changing a single character of x. By considering the worst case, it is clear that such a change would be limited to kM for k character changes. In the case of the DNA alphabet this value of kM reduces to $3k$.

Definition 3.6 (Absolute Character Difference Sum Filter)**.** Using the num notation as defined in Section 3.2.1, we can define the absolute character difference sum filter F_2 on a string x of length m as:

$$F_2(x) = \Big(\sum_{i=0}^{m-2} |x_N[i] - x_N[i+1]| \Big) + |x_N[m-1] - x_N[0]| \quad F_2 : \Sigma^* \mapsto \mathbb{N}$$

This filter takes all pairs of consecutive characters of the string including the final and first characters. The filter then calculates the absolute difference of each pair and finds the final total of all the absolute differences. With respect to the problem defined in Section 3.2.2, we can rule out a match with a window starting at position i of the text t where the following does not hold:

$$|F_2(t[i\,..\,i+m-1]) - F_2(x)| \leq 2kM$$

The constant $2M$ represents double the maximum possible absolute change in the value of $F_2(S)$ induced by changing a single character of x. By considering the worst case, it is clear that such a change would be limited to $2kM$ for k character changes. In the case of the DNA alphabet this value of $2kM$ reduces to $6k$.

Definition 3.7 (Individual Character Count Filter)**.** For this filter we map to an integer vector of length $|\Sigma|$, the size of the alphabet under consideration. Each entry in the vector is associated with a specific character from the alphabet:

$$F_3(x) = \begin{bmatrix} |\{x[i] = \Sigma[0] : 0 \leq i \leq m-1\}| \\ |\{x[i] = \Sigma[1] : 0 \leq i \leq m-1\}| \\ |\{x[i] = \Sigma[2] : 0 \leq i \leq m-1\}| \\ \vdots \\ |\{x[i] = \Sigma[|\Sigma|-1] : 0 \leq i \leq m-1\}| \end{bmatrix}$$

Recall that in Section 2.2 we defined the notation $\Sigma[i]$ for some i. The filter enumerates the number of occurrences of each character in the pattern and stores the result as a vector. This structure is known in the literature as a Parikh Vector and will be further utilised under the same definition in Chapter 4 [66]. In the case of the DNA alphabet this will be a vector of size 4 corresponding to number of occurrences of A, C, G, T. With respect to the problem defined in Section 3.2.2, we can rule out a match with a window starting at position i of the text t where the following does not hold:

$$F_3(t[i\,..\,i+m-1])[j] - F_3(x)[j] \leq k \;\; \forall j \in \{0, \ldots, |\Sigma|-1\}$$

Each of these filters can be run on the full text t of length n in linear time as an initial preprocessing stage. With each filter, the search space may be significantly reduced. Once the search space is reduced, a final verification step occurs on the remaining text windows to determine the final set of match positions.

3.3.2 Algorithm

The algorithm we use is implemented in 2 stages, these being the filtering stage and the verification stage. In the filtering stage, the text search space is massively reduced in total time $\mathcal{O}(m+n)$ where m is the length of the pattern and n is the length of the text. In the verification stage, the text windows in the reduced search space are individually tested for match against the pattern. This stage is an involved process, taking advantage of partitioning techniques and the Knuth Morris Pratt (KMP) search algorithm [67].

The pseudo-code shown takes only a single filter as input, though in practice we use our 3 previously defined filters, and the algorithm may also be extended appropriately to accommodate any number of filters. We introduce several subroutines, which must be implemented accordingly for each individual filter. These are as follows:

$\mathbf{F_{lower}}(p, k)$

For a given filter F, calculates the *lowest* value that any text window could map to without being disqualified as a potential k-match against a given pattern p. This will depend on the filter F, the pattern p and the approximation k.

$\mathbf{F_{upper}}(p, k)$

For a given filter F, calculates the *highest* value that any text window could map to without being disqualified as a potential k-match against a given pattern p. This will depend on the filter F, the pattern p and the approximation k.

$\mathbf{F_{next}}(t, i, f)$

For a given filter F, returns the filter map of the window at position $i+1$ of a text t, namely $F(t[i+1, \ldots, i+m])$. The required inputs are the text t, the index immediately preceding the desired text window, namely i, and the filter map of the immediately preceding text window, namely $f = F(t[i, \ldots, i+m-1])$. This function performs more efficiently than applying the filter $F(t[i+1, \ldots, i+m])$ directly, by using rolling calculation.

minError(p, t)

This subroutine is independent of any filters used and performs the primary logic of the verification step. Given a pattern p of length m and a text window t, this subroutine calculates the minimum error difference (Hamming distance) across all rotations of the pattern p against the text window t.

3.4 Pseudo-code

Function CIRCULARSTRINGSEARCH(t, n, p, m, k, F)

input :

t = Text to be searched.
n = Length of the text t.
p = Circular pattern to be found.
m = Length of the circular pattern p.
k = Maximum error permissible in reported matches.
F = Filter function.

output:

A set of all pairs (i, e) such that some rotation of the pattern p matches the text t at index i with an error of $e \leq k$.

```
candidateWindows ← LinkedList()
verifiedWindows ← LinkedList()
f ← F(t[0, ..., m − 1])
lowerBound ← F_LOWER(p, k)
upperBound ← F_UPPER(p, k)
for i ← 0 to n − m do
    if lowerBound ≤ f ≤ upperBound then
        candidateWindows.push(i)
    if i < n − m then
        f = F_NEXT(t, i, f)
foreach i in candidateWindows do
    e = minError(p, t[i, ..., i + m − 1])
    if e ≤ k then
        verifiedWindows.push((i, e))
return verifiedWindows
```

The pseudo-code demonstrates an overview of the circular string search logic. In practice this algorithm was implemented in JavaScript with 3 stages of filtering and includes additionally intricacy beneath the high-level view. Details of the underlying code in subroutines is omitted. The precise methods can be understood by consulting the original source code available at the following link:

https://github.com/steven31415/approximate-circular-pattern-matching

3.5 Implementation

3.5.1 Interface

A GUI was developed to enable interaction with the algorithm. This provides a basic interface for submitting pattern and text queries for the non-technical user. Parameters can be set by various components of the interface, and the results exported. Note that the only data transmitted via the network when interacting with this interface is the HTML and JavaScript code itself and optionally the output data.

This method of running the executable code on the host's machine is an often used technique, when requiring a large amount of data to be processed, but with a small amount of resulting data [68]. The traditional method of web-based data processing, involves a transfer of the input data to the web server, where processing takes place remotely, and results are transferred back to the user. With this implementation, we instead take the approach of transferring the code describing the algorithm to the host, where it processes the data and displays the results in the browser. The potentially large input data never moves across the network.

The full interface is available for immediate use at the following URL:

https://nms.kcl.ac.uk/steven.watts/acss/acss.html

3.5.2 Performance Analysis

When we carried out performance testing, both the client-side and server-side components of the system were tested within the same computing environment, disconnected from any networks in order to prevent interference. The machine used was running `Ubuntu 16.04 64-Bit` with an `Intel Core i5-6600K CPU` and `2 x 8GB DDR3 RAM`.

Initial tests intended to determine average run-time and average search space reductions when applying the filtering process and were performed on randomly generated data. Random numbers used during testing were obtained from `http://random.org`, providing a source of true random numbers, accessible via an API [69]. For a given string length, we randomly generated data by producing a random sequence over the 4-letter DNA alphabet, following a uniform distribution, using true random numbers. Throughout testing, whenever multiple tests were performed with the same configuration of input parameters, a new pair of random pattern and text sequences was automatically generated for each test.

We performed testing across multiple text sizes, pattern sizes and approximation levels k. Within the performance testing data, string sizes are measured by the number of bytes required for storage, with a single DNA character being equivalent to 1 byte with an 8-bit ASCII encoding [70]. The number of seconds taken for the filtering algorithm to execute, is referred to as the execution time. The execution time includes only the time taken to perform the filtering stage of the algorithm, and excludes the time taken to verify potential match locations. This is to ensure the potential utility of the filtering method is studied in isolation. Note that when testing this implementation, the transfer time of query data across the network is taken to be effectively zero. These initial results are shown in Fig. 3.8 and Fig. 3.9.

		Text Size (MB)					
Pattern Size (kB)	Permitted Error k	250		500		1000	
1	0	n	20	n	10	n	5
		μ	96.9 s	μ	192.3 s	μ	385.4 s
		σ	2.9 s	σ	2.0 s	σ	4.4 s
	1	n	20	n	10	n	5
		μ	98	μ	196	μ	388.7 s
		σ	3.2	σ	3.7	σ	7.2 s
	2	n	20	n	10	n	5
		μ	98.8 s	μ	197.6 s	μ	394.5 s
		σ	1.9 s	σ	2.5 s	σ	10.2 s
	3	n	20	n	10	n	5
		μ	100.4 s	μ	201.3 s	μ	391.3 s
		σ	2.4 s	σ	3.7 s	σ	5.9 s
10	0	n	20	n	10	n	5
		μ	94.9 s	μ	190.4 s	μ	382.7 s
		σ	1.2 s	σ	1.7 s	σ	1.3 s
	1	n	20	n	10	n	5
		μ	96	μ	192	μ	387.6 s
		σ	0.9	σ	2.3	σ	8.1 s
	2	n	20	n	10	n	5
		μ	96.4 s	μ	193.2 s	μ	385.1 s
		σ	1.2 s	σ	1.9 s	σ	7.2 s
	3	n	20	n	10	n	5
		μ	97.9 s	μ	192.4 s	μ	390.5 s
		σ	4.3 s	σ	2.8 s	σ	6.8 s
100	0	n	20	n	10	n	5
		μ	127.0 s	μ	251.3 s	μ	504.3 s
		σ	3.4 s	σ	1.4 s	σ	2.7 s
	1	n	20	n	10	n	5
		μ	127	μ	252	μ	508.0 s
		σ	2.6	σ	3.1	σ	10.6 s
	2	n	20	n	10	n	5
		μ	126.5 s	μ	252.8 s	μ	509.9 s
		σ	2.9 s	σ	1.7 s	σ	39.6 s
	3	n	20	n	10	n	5
		μ	126.7 s	μ	246.8 s	μ	506.7 s
		σ	2.7 s	σ	11.1 s	σ	6.4 s

FIGURE 3.8: **Average run-time** in seconds to complete the filtering process for various input configurations. n is number of tests, μ is average run-time, σ is standard deviation. Both pattern and text are randomly generated.

		Text Size (MB)					
Pattern Size (kB)	Permitted Error k	250		500		1000	
1	0	n	20	n	10	n	5
		μ	100.0000%	μ	100.0000%	μ	100.0000%
		σ	0.0000%	σ	0.0000%	σ	0.0000%
	1	n	20	n	10	n	5
		μ	99.9977%	μ	99.9970%	μ	99.9980%
		σ	0.0026%	σ	0.0014%	σ	0.0020%
	2	n	20	n	10	n	5
		μ	99.9788%	μ	99.9717%	μ	99.9681%
		σ	0.0187%	σ	0.0200%	σ	0.0319%
	3	n	20	n	10	n	5
		μ	99.9068%	μ	99.8594%	μ	99.9666%
		σ	0.0857%	σ	0.0835%	σ	0.0324%
10	0	n	20	n	10	n	5
		μ	100.0000%	μ	100.0000%	μ	100.0000%
		σ	0.0000%	σ	0.0000%	σ	0.0000%
	1	n	20	n	10	n	5
		μ	100.0000%	μ	100.0000%	μ	100.0000%
		σ	0.0000%	σ	0.0000%	σ	0.0000%
	2	n	20	n	10	n	5
		μ	99.9997%	μ	99.9997%	μ	99.9999%
		σ	0.0003%	σ	0.0003%	σ	0.0000%
	3	n	20	n	10	n	5
		μ	99.9990%	μ	99.9992%	μ	99.9983%
		σ	0.0009%	σ	0.0007%	σ	0.0014%
100	0	n	20	n	10	n	5
		μ	100.0000%	μ	100.0000%	μ	100.0000%
		σ	0.0000%	σ	0.0000%	σ	0.0000%
	1	n	20	n	10	n	5
		μ	100.0000%	μ	100.0000%	μ	100.0000%
		σ	0.0000%	σ	0.0000%	σ	0.0000%
	2	n	20	n	10	n	5
		μ	100.0000%	μ	100.0000%	μ	100.0000%
		σ	0.0000%	σ	0.0000%	σ	0.0000%
	3	n	20	n	10	n	5
		μ	100.0000%	μ	100.0000%	μ	100.0000%
		σ	0.0000%	σ	0.0000%	σ	0.0000%

FIGURE 3.9: **Average search space reduction** of the original text as a percentage of original size, for various input configurations. n is number of tests, μ is average search space reduction, σ is standard deviation. Both pattern and text are randomly generated.

From the data collected, the main influence on the run-time of the algorithm was the size of the text to be searched, appearing to have a linear relationship within this sample, as should be expected. However the run-time increases sub-linearly with an increase in pattern size. Varying the permitted error k appears to have a minor influence on the run-time.

Due to the relatively large size of the patterns tested, the search space may be reduced by almost 100% in each instance. However, within these tests it appears that this is primarily influenced by the permitted error k. The influence that input parameters have on search space reduction may be more easily seen within the data produced in Fig. 3.10, where relatively smaller pattern sizes were used in testing.

Fig. 3.10 was produced by performing an additional series of tests, comparing the results for two identically sized text patterns, sourced from real and random data sources respectively. Specifically, the random data sample was produced in the same manner as described for the tests on randomly generated texts within Fig. 3.8 and Fig. 3.9. The real data sample was taken as a random substring of DNA from an HIV source [71]. The specific sample of real data used may be found at `https://raw.githubusercontent.com/steven31415/approximate-circular-pattern-matching/master/HIV_sample.txt`.

This comparison between random and real data was performed in order to explore the possibility that the structure of biological data varies from randomly distributed data in such a way as to significantly alter the performance of the algorithm.

Some minor performance differences were observed between real and random text, though the order of complexity appeared to remain unchanged.

		Run-Time Tests				Search Space Reduction Tests			
Pattern Size (B)	Permitted Error k	Real Data		Random Data		Real Data		Random Data	
5	0	n	50	n	50	n	50	n	50
		μ	62.9 s	μ	63.3 s	μ	98.7144%	μ	98.4246%
		σ	9.7 s	σ	8.9 s	σ	0.6972%	σ	0.8707%
	1	n	50	n	50	n	50	n	50
		μ	72	μ	71	μ	70.0676%	μ	68.7340%
		σ	9.3	σ	8.2	σ	13.4514%	σ	13.7712%
	2	n	50	n	50	n	50	n	50
		μ	89.3 s	μ	90.1 s	μ	26.7360%	μ	19.2182%
		σ	9.6 s	σ	9.6 s	σ	17.5758%	σ	14.0940%
	3	n	50	n	50	n	50	n	50
		μ	98.3 s	μ	95.6 s	μ	6.8898%	μ	6.0382%
		σ	9.2 s	σ	9.6 s	σ	7.6994%	σ	9.1541%
10	0	n	50	n	50	n	50	n	50
		μ	62.8 s	μ	62.1 s	μ	99.8158%	μ	99.7538%
		σ	6.3 s	σ	6.5 s	σ	0.1497%	σ	0.1925%
	1	n	50	n	50	n	50	n	50
		μ	67	μ	68	μ	90.3458%	μ	86.9742%
		σ	7.0	σ	7.1	σ	6.6676%	σ	7.4289%
	2	n	50	n	50	n	50	n	50
		μ	73.7 s	μ	81.1 s	μ	55.9508%	μ	54.4920%
		σ	7.1 s	σ	13.0 s	σ	17.2080%	σ	19.3734%
	3	n	50	n	50	n	50	n	50
		μ	89.9 s	μ	91.5 s	μ	30.1900%	μ	25.2692%
		σ	13.4 s	σ	9.4 s	σ	20.0760%	σ	18.7837%
100	0	n	50	n	50	n	50	n	50
		μ	61.7 s	μ	61.5 s	μ	100.0000%	μ	99.9968%
		σ	6.8 s	σ	7.7 s	σ	0.0000%	σ	0.0082%
	1	n	50	n	50	n	50	n	50
		μ	63	μ	63	μ	99.9052%	μ	99.7538%
		σ	7.3	σ	9.8	σ	0.1165%	σ	0.2084%
	2	n	50	n	50	n	50	n	50
		μ	62.2 s	μ	63.3 s	μ	99.5892%	μ	98.5390%
		σ	6.8 s	σ	7.3 s	σ	0.4201%	σ	1.2438%
	3	n	50	n	50	n	50	n	50
		μ	64.7 s	μ	67.5 s	μ	98.4228%	μ	94.8126%
		σ	5.8 s	σ	7.6 s	σ	1.1786%	σ	3.9438%

FIGURE 3.10: Comparison of average run-time and average search space reduction for both real text and randomly generated text, for various input configurations. n is number of tests, μ is average search space reduction, σ is standard deviation. The size of both the real text sample and randomly generated text used across all tests was 10kB.

3.6 Conclusion

After both theoretical and experimental analysis of the web tool, it would appear it has succeeded in satisfying the initial target of performing Approximate Circular Pattern Matching within reasonable execution time and without need for significant data transfer.

As the pattern size approaches the text size, the algorithm performs linearly. However for smaller patterns execution time is still within reasonable bounds. The analysis shows that the conclusions made with regard to performance on random data will also apply to real world data and provide a practical solution to real matching problems. This practical approach is bolstered by the fact that no data transfer is required, removing the need to satisfy data caps that may be required by other online tools. Additionally no significant technical expertise is required to apply the implementation, with a publicly accessible and simple interface available to end users.

Further work in this area could see further string processing tools being implemented on the server-side in a similar fashion. This would be particularly useful for the many algorithms which would ordinarily struggle to handle large volumes of data in their current implementations. Our methods currently serve as a proof of concept, and could benefit from further improvement, particularly with regard to accepting data that may be encountered in practice. This may include expanding on the DNA characters that may be accepted as input, and further allowing for the various standardised DNA streams such as FASTA, FASTQ, EMBL or others.

The implementation and associated source code is made publicly available via the links provided in Section 3.5.1, in the hope that it may prove beneficial to those seeking to create further improvements within the domain of circular string search.

Chapter 4

Abelian Palindromes

A string is called a *palindrome* if it reads the same from left to right. In this chapter we define the new concept of an *abelian palindrome* which satisfies the property of being abelian equivalent to some palindrome of the same length. The identification of abelian palindromes presents a novel combinatorial problem, with potential applications in filtering strings for palindromic factors. This may be seen as similar to the concept of filtering as used in Chapter 3. This chapter presents an algorithm to efficiently identify abelian palindromes, and additionally generate an *abelian palindromic array*, indicating the longest abelian palindrome at each location. Specifically, for an alphabet of size $|\Sigma| \leq \log_2(n)$ and after $\mathcal{O}(n)$ time preprocessing using $\mathcal{O}(n + |\Sigma|)$ space, we may determine if any factor is abelian palindromic in $\mathcal{O}(1)$ time. Additionally, we may determine the abelian palindromic array in $\mathcal{O}(|\Sigma|n)$ time. A further specification may be made on the algorithmic complexity when this condition on alphabet size $|\Sigma|$ is relaxed.

4.1 Background

The identification of palindromic factors in strings, has been a much studied area of stringology, due to the interesting combinatorial aspects and the strong ties with genetic analysis, where palindromes often correspond to significant structures in DNA [72].

Variations of the palindrome identification problem have been frequently introduced, for example Karhumäki et al. presented results on k-abelian palindromes on rich and poor words [73]. Holub et al. considered the problem as applied to binary words, investigating the properties of palindromic factors of binary strings [74].

We introduce our own simple modification to the problem, yet to be explored, namely abelian palindromes. Though an interesting combinatorial problem in it itself, an efficient method of detecting abelian palindromes can potentially provide a filter by which ordinary palindromic factors may be deduced. This follows from the fact that an ordinary palindrome must necessarily also be an abelian palindrome, and therefore the search space may be reduced if non abelian palindromes can be efficiently dismissed.

Likewise, the abelian palindromic array may potentially be used to assist in the calculation of the ordinary palindromic array, for the purpose of performing a greedy factorisation of a string into ordinary palindromes. This follows from the fact that the abelian palindromic array provides an upper bound for the equivalent value in the ordinary palindromic array.

4.2 Problem Outline

4.2.1 Terminology

We denote the *reverse* string of x by x^R as the string obtained when reading x from right to left, i.e. $x^R = x[n-1]x[n-2]\dots x[1]x[0]$. We say a string x is a *palindrome* when $x = x^R$.

We make use of the bit-wise *exclusive or* (XOR) operation between two binary strings x and y of the same length $|x| = |y|$, denoted $x \oplus y$. This adheres to the standard definition of XOR on two binary strings, i.e. $z[i] = (x[i] + y[i]) \pmod 2$ where $z = x \oplus y$. We may similarly apply the XOR operation to integers, $x, y \in \mathbb{Z}$ by converting x and y to their respective binary equivalents, performing the XOR operation, and converting the binary result into an integer. For example, given $x = 5, y = 11$ we have $x \oplus y = 5 \oplus 11 = \texttt{0101} \oplus \texttt{1011} = \texttt{1110} = 14$.

The concept of abelian strings relates to the idea of disregarding the order of appearance of characters in a string, and concerning ourselves only with the number of occurrences of each character within the string. With this in mind, we wish to define the concept of an *abelian palindrome*. To facilitate this, we must first refer to the definition of a *Parikh Vector* [66].

Definition 4.1. The *Parikh vector* $\mathcal{P}(t)$ of a string t over the alphabet Σ, is a vector of size $|\Sigma|$ which enumerates the number of occurrences of each character of the alphabet in t. If the character $\sigma \in \Sigma$ has ordinality $i = \text{ORD}(\sigma)$ in the lexicographical ordering of the alphabet Σ, then $\mathcal{P}(t)[i]$ stores the number of occurrences of σ in t.

We say that two strings t_1 and t_2 are *abelian equivalent* denoted $t_1 \approx_a t_2$ if and only if they have the same Parikh vector, i.e. are permutations of each other.

For example, the string $t_1 =$ `accgta` has the Parikh vector $\mathcal{P}(t_1) = (2, 2, 1, 1)$. The string $t_2 =$ `gactca` has the same Parikh vector and thus $t_1 \approx_a t_2$. We may now define the concept of an abelian palindrome.

Definition 4.2. A string t is an *abelian palindrome* if and only if there exists some palindrome p such that $p \approx_a t$.

Note that in general, a string t will more easily satisfy the abelian palindromic property over the palindromic property. This comes as a direct result of Lem. 4.3, which follows clearly from Def. 4.2.

Lemma 4.3.

$$t \text{ palindromic} \implies t \text{ abelian palindromic} \tag{4.1}$$

$$t \text{ not abelian palindromic} \implies t \text{ not palindromic} \tag{4.2}$$

Proof. Assume t is palindromic, choose $p = t$. Therefore we have $p = t \approx_a t$. Thus t is abelian palindromic and Statement 4.1 is proven. Statement 4.2 follows as the contrapositive of Statement 4.1. □

4.2.2 Problem Statements

We now formally define the two related problems addressed within this chapter, namely Abelian Palindromic Factor Recognition and the Abelian Palindromic Array:

Abelian Palindromic Factor Recognition

Input:
A string t of length n.

Output:
A function F : $\{0\,..\,n-1\} \times \{0\,..\,n-1\} \to \{\texttt{true}, \texttt{false}\}$ where $F(i, j)$ returns `true` if $t[i\,..\,j]$ is abelian palindromic and `false` if $t[i\,..\,j]$ is not abelian palindromic, in $\mathcal{O}(1)$ time.

ABELIAN PALINDROMIC ARRAY

Input:
A string t of length n.

Output:
An array A of size n such that $A[i]$ stores the length of the longest abelian factor of t occurring at position i, i.e. as a prefix of $t[i\,..\,n-1]$.

Note that recognising a palindromic factor is naturally easier than generating the abelian palindromic array, by virtue of the fact that the former is a verification problem whereas the latter is not.

4.3 Solution

4.3.1 Tools

We wish to efficiently identify abelian palindromic factors within a string. To enable this, we define some further concepts and auxiliary data structures.

From Def. 4.2, it is clear that whether a string t is an abelian palindrome is dependant on the values in its Parikh vector $\mathcal{P}(t)$, specifically the number of values that are odd or even. We use $|\mathcal{P}(t)|$ to refer to the total number of values in $\mathcal{P}(t)$, and further use $|(\mathcal{P}(t))|_{\texttt{odd}}$ and $|(\mathcal{P}(t))|_{\texttt{even}}$ to refer to the number of odd and even values in $\mathcal{P}(t)$ respectively. This notation allows us to succinctly describe the defining quality of an abelian palindrome in Lem. 4.4.

Lemma 4.4. *t abelian palindromic* $\iff 0 \leq |(\mathcal{P}(t))|_{\texttt{odd}} \leq 1$.

Proof. We refer to the length of t as n. We call $t_l = t[0\,..\,\lfloor\frac{n-1}{2} - 1\rfloor]$ the *left half* of t and $t_r = t[\lceil\frac{n-1}{2} + 1\rceil\,..\,n-1]$ the *right half* of t. Note that if n is even, $t = t_l\,t_r$. If n is odd, $t = t_l\,c\,t_r$ where $c = t[\frac{n-1}{2}]$. For an ordinary palindrome p, it is clear that if a character occurs m times in p_l it must correspondingly occur m times in p_r, to preserve the palindromic property of p.

We first show that if t is an abelian palindrome, the number of odd values in the Parikh vector $|(\mathcal{P}(t))|_{\texttt{odd}}$ can not exceed 1 by contradiction. Let us assume that $|(\mathcal{P}(t))|_{\texttt{odd}} > 1$. In this case, we have at least 2 different characters $\sigma_1, \sigma_2 \in \Sigma$ with an odd number of occurrences in t. For any permutation of the characters in t, at least one of these two characters must have all its occurrences contained entirely within t_l and t_r, and we call

this character σ. The character σ therefore occurs $2m$ times in t where m is the number of occurrences of σ in t_l. Therefore σ has an even number of occurrences, which leads us to a contradiction. Therefore we conclude that $0 \leq |(\mathcal{P}(t))|_{\texttt{odd}} \leq 1$.

We now show that we can always form a palindrome from a permutation of t when $0 \leq |(\mathcal{P}(t))|_{\texttt{odd}} \leq 1$. In the notation below we use p_l^R to represent the reversal of p_l.

Let us assume $|(\mathcal{P}(t))|_{\texttt{odd}} = 0$. In this case, $\mathcal{P}(t)$ contains only even values. We distribute the characters evenly to form an even length palindrome p such that $p \approx_a t$ as follows (with braces under characters indicating the number of repetitions of that character):

$$p_l = \underbrace{\Sigma[0]}_{\frac{1}{2}\mathcal{P}(t)[0]} \underbrace{\Sigma[1]}_{\frac{1}{2}\mathcal{P}(t)[1]} \ldots \underbrace{\Sigma[|\Sigma|-1]}_{\frac{1}{2}\mathcal{P}(t)[|\Sigma|-1]}$$

$$p = p_l \; p_l^R$$

Now let us assume $|(\mathcal{P}(t))|_{\texttt{odd}} = 1$. In this case, $\mathcal{P}(t)$ contains a single odd entry corresponding to some character $c = \Sigma[i]$. We distribute the characters evenly, placing the character c at the centre, to form an odd length palindrome p such that $p \approx_a t$ as follows:

$$p_l = \underbrace{\Sigma[0]}_{\frac{1}{2}\mathcal{P}(t)[0]} \ldots \underbrace{\Sigma[i-1]}_{\frac{1}{2}\mathcal{P}(t)[i-1]} \underbrace{\Sigma[i+1]}_{\frac{1}{2}\mathcal{P}(t)[i+1]} \ldots \underbrace{\Sigma[|\Sigma|-1]}_{\frac{1}{2}\mathcal{P}(t)[|\Sigma|-1]}$$

$$p = p_l \; \Sigma[i] \; p_l^R$$

Thus we have shown that $0 \leq |(\mathcal{P}(t))|_{\texttt{odd}} \leq 1$ is both a necessary and sufficient condition for t to be an abelian palindrome. Therefore Lem. 4.4 follows. □

To illustrate the application of Lem. 4.4, we present examples for even length, odd length and non abelian palindromes respectively:

$$
\begin{aligned}
& t_1 = \texttt{GTAGAGCGCATA} \\
\implies & \mathcal{P}(t_1) = (4, 2, 4, 2) \\
\implies & |(\mathcal{P}(t))|_{\texttt{odd}} = 0 \\
\implies & t_1 \approx_a \texttt{AACGGTTGGCAA}
\end{aligned}
$$

$$
\begin{aligned}
& t_2 = \texttt{GGTTAGCTATG} \\
\implies & \mathcal{P}(t_2) = (2, 1, 4, 4) \\
\implies & |(\mathcal{P}(t))|_{\texttt{odd}} = 1 \\
\implies & t_2 \approx_a \texttt{AGGTTCTTGGA}
\end{aligned}
$$

$$
\begin{aligned}
& t_3 = \texttt{AGCGATATGACT} \\
\implies & \mathcal{P}(t_3) = (4, 2, 3, 3) \\
\implies & |(\mathcal{P}(t))|_{\texttt{odd}} = 2 \\
\implies & t_3 \text{ not abelian palindromic}
\end{aligned}
$$

The next aim is to describe a new data structure that will prove useful in recognising palindromic factors, beginning with some new definitions.

Lem. 4.4 provides us with a useful criterion by which we can seek longest abelian palindromes. We first provide some additional definitions which will prove useful:

Definition 4.5. A *prefix Parikh vector* $\mathcal{P}_i(t)$ of a string t is the Parikh vector of the ith prefix of t:

$$\mathcal{P}_i(t) = \mathcal{P}(t[0\,..\,i]) \text{ for } 0 \leq i \leq n-1$$

Definition 4.6. A *parity vector* $\mathbb{P}(t)$ of a string t over the alphabet Σ is a bit vector of length Σ which indicates the parity (even or odd) of the number of occurrences of each character of Σ in t (0 indicates `even`, 1 indicates `odd`):

$$\mathbb{P}(t)[i] = \mathcal{P}(t)[i] \pmod 2$$

Definition 4.7. A *prefix parity vector* $\mathbb{P}_i(t)$ of a string t is the parity vector of the ith prefix of t:

$$\mathbb{P}_i(t) = \mathbb{P}(t[0\,..\,i]) \text{ for } 0 \leq i \leq n-1$$

Definition 4.8. A *parity integer* $\hat{\mathbb{P}}(t)$ of a string t over the alphabet Σ is a decimal integer representing the value of the parity vector $\mathbb{P}(t)$ when interpreted as a binary number, with the order of magnitude of each bit determined by the lexicographical order of the alphabet Σ:

$$\hat{\mathbb{P}}(t) = \sum_{i=0}^{|\Sigma|-1} 2^i \times \mathbb{P}(t)[i]$$

Definition 4.9. A *prefix parity integer* $\hat{\mathbb{P}}_i(t)$ of a string t is the parity integer of the ith prefix of t:

$$\hat{\mathbb{P}}_i(t) = \hat{\mathbb{P}}(t[0\,..\,i]) \text{ for } 0 \leq i \leq n-1$$

Definition 4.10. The *prefix parity integer array* $\hat{\mathbb{P}}_A(t)$ of a string t of length n is an integer array of length n, which contains the value of $\hat{\mathbb{P}}_i(t)$ at each position i:

$$\hat{\mathbb{P}}_A(t)[i] = \hat{\mathbb{P}}_i(t)$$

The prefix parity integer array $\hat{\mathbb{P}}_A(t)$ (example: see bottom of Fig. 4.11) is the key to identifying longest abelian palindromes in a string t. To observe this, we note that the Parikh vector of a factor of t can be determined by evaluating the difference between the two prefix Parikh vectors at the start and end indexes of the factor. The parity vector and parity integer of a factor can also be determined in a similar way, by employing the bit-wise exclusive or (XOR) operation. We summarise these observations in Lem. 4.12.

Lemma 4.12. *Given a string t:*

$$\mathcal{P}(t[i\,..\,j]) = \mathcal{P}_j(t) - \mathcal{P}_{i-1}(t) \tag{4.1}$$

$$\mathbb{P}(t[i\,..\,j]) = \mathbb{P}_j(t) \oplus \mathbb{P}_{i-1}(t) \tag{4.2}$$

$$\hat{\mathbb{P}}(t[i\,..\,j]) = \hat{\mathbb{P}}_j(t) \oplus \hat{\mathbb{P}}_{i-1}(t) \tag{4.3}$$

Proof. Given a factor $f = t[i\,..\,j]$ we have $t[0\,..\,j] = t[0\,..\,i-1]\,f$. Therefore it follows that $\mathcal{P}(t[0\,..\,j]) = \mathcal{P}(t[0\,..\,i-1]) + \mathcal{P}(f) \implies \mathcal{P}(f) = \mathcal{P}(t[0\,..\,j]) - \mathcal{P}(t[0\,..\,i-1])$. Thus Statement 4.1 is proven.

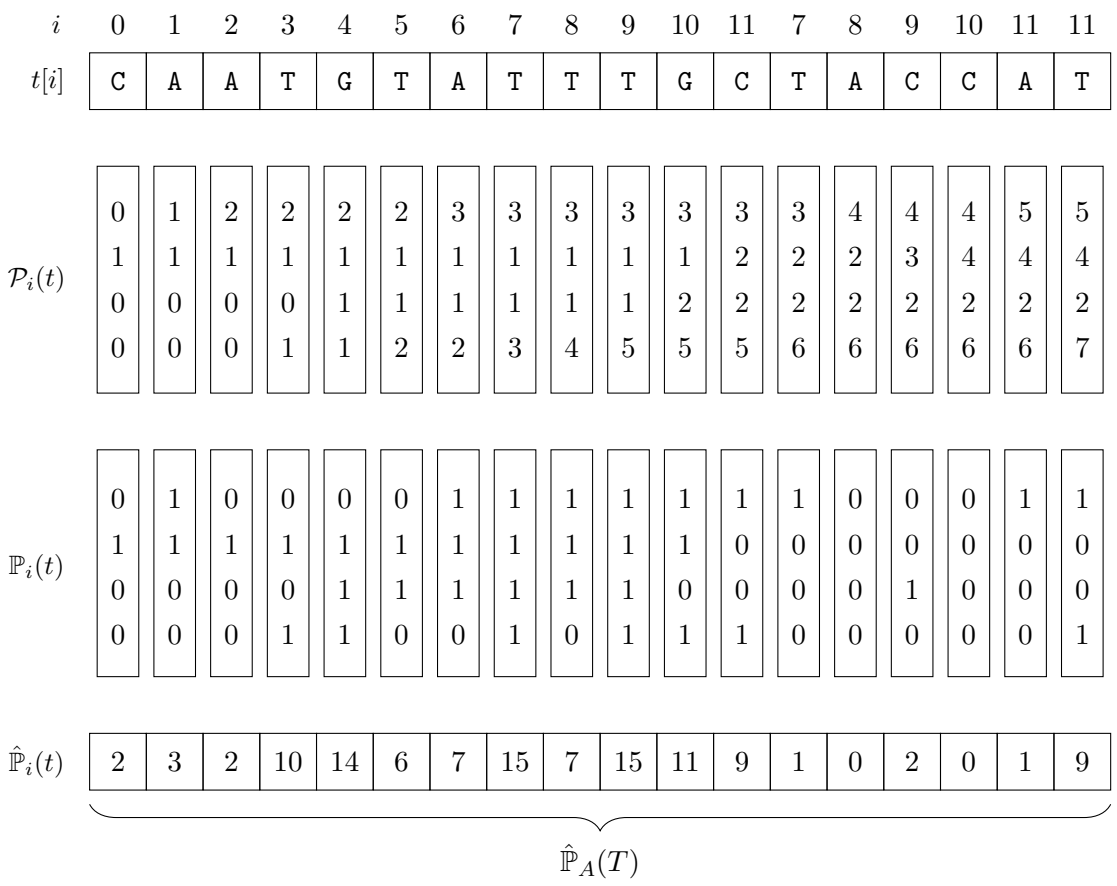

FIGURE 4.11: Example of prefix Parikh vectors, prefix parity vectors and prefix parity integers for the string $t = \texttt{caatgtatttgctaccat}$.

Statement 4.2 follows from Statement 4.1 by observing that the truth table for the XOR operator is analogous to the parity table for the subtraction operator (when we interpret 0 as `even`, 1 as `odd`). Note that subtraction (mod 2) and XOR are both commutative operations, and therefore the order of inputs is unimportant for both.

Statement 4.3 is simply an alternative formulation of Statement 4.2 in the form of parity integers instead of parity vectors. □

Given $\hat{\mathbb{P}}_A(t)$, it now becomes simple to verify whether a factor $t[i\,..\,j]$ is an abelian palindrome, i.e. $0 \leq |\mathcal{P}(t[i\,..\,j])|_{\texttt{odd}} \leq 1$.

Lemma 4.13. *Given a text t over the alphabet Σ, the following holds:*

$$t[i\,..\,j] \text{ abelian palindromic} \iff \hat{\mathbb{P}}_j(t) \oplus \hat{\mathbb{P}}_{i-1}(t) \in \{0\} \cup \{2^0, 2^1, \ldots, 2^{|\Sigma|-1}\}$$

Proof. The lemma follows from the application of previously defined lemmas. We use brackets under runs of characters to indicate the length of that run of characters:

$$
\begin{aligned}
& t[i\,..\,j] \text{ is abelian palindromic} \\
\underset{\text{Lem. 4.4}}{\iff} \; & 0 \le |\mathcal{P}(t[i\,..\,j])|_{\texttt{odd}} \le 1 \\
\iff \; & |\mathcal{P}(t[i\,..\,j])|_{\texttt{odd}} = 0 \;\vee\; |\mathcal{P}(t[i\,..\,j])|_{\texttt{odd}} = 1 \\
\underset{\text{Def. 4.6}}{\iff} \; & \mathbb{P}(t[i\,..\,j]) = \underbrace{0\ldots0}_{|\Sigma|} \vee \mathbb{P}(t) \in \{\underbrace{0\ldots0}_{|\Sigma|-1}1\,,\,\underbrace{0\ldots0}_{|\Sigma|-2}10\,,\,\ldots\,,\,1\underbrace{0\ldots0}_{|\Sigma|-1}\} \\
\underset{\text{Def. 4.8}}{\iff} \; & \mathbb{P}(t[i\,..\,j]) = 0 \;\vee\; \hat{\mathbb{P}}(t) \in \{2^0, 2^1, 2^2, \ldots, 2^{|\Sigma|-1}\} \\
\underset{\text{Lem. 4.12}}{\iff} \; & \hat{\mathbb{P}}_j(t) \oplus \hat{\mathbb{P}}_{i-1}(t) \in \{0\} \cup \{2^0, 2^1, \ldots, 2^{|\Sigma|-1}\}
\end{aligned}
$$

□

Lem. 4.13 immediately leads us to Lem. 4.14, which allows us to identify the longest factor of a string t starting at i which is abelian palindromic.

Lemma 4.14. *Given a text t over the alphabet Σ, the longest abelian palindromic factor of t occurring at position i is $t[i\,..\,j]$ where j satisfies the following:*

$$j = \max\{j' : \hat{\mathbb{P}}_{j'}(t) \in M(t,i)\}$$

$$M(t,i) = \{\hat{\mathbb{P}}_{i-1}(t) \oplus k : k \in \{0\} \cup \{2^0, 2^1, \ldots, 2^{|\Sigma|-1}\}\}$$

For a given string t and position i, we call $M(t,i)$ the match set*.*

Proof. For a fixed i, the longest $t[i\,..\,j]$ which is abelian palindromic is found by determining the largest j, such that i and j satisfy the condition in Lem. 4.13. We may derive the match set $M(t,i)$ from this condition by employing the fact that XOR is

commutative:

$$\begin{aligned}
&\hat{\mathbb{P}}_{j'}(t) \oplus \hat{\mathbb{P}}_{i-1}(t) \in \{0\} \cup \{2^0, 2^1, \ldots, 2^{|\Sigma|-1}\} \\
\iff\ &\exists k \in \{0\} \cup \{2^0, 2^1, \ldots, 2^{|\Sigma|-1}\} \text{ such that } \hat{\mathbb{P}}_{j'}(t) \oplus \hat{\mathbb{P}}_{i-1}(t) = k \\
\iff\ &\exists k \in \{0\} \cup \{2^0, 2^1, \ldots, 2^{|\Sigma|-1}\} \text{ such that } \hat{\mathbb{P}}_{j'}(t) = \hat{\mathbb{P}}_{i-1}(t) \oplus k \\
\iff\ &\mathbb{P}_{j'}(t) \in \{\hat{\mathbb{P}}_{i-1}(t) \oplus k : k \in \{0\} \cup \{2^0, 2^1, \ldots, 2^{|\Sigma|-1}\}\} = M(t,i)
\end{aligned}$$

Thus for a given i, the largest j' satisfying the above condition gives us the j corresponding to the largest abelian palindromic factor $t[i\,..\,j]$. □

At this stage it now becomes useful to define a simple data structure that will prove useful for identifying longest abelian palindromes.

Definition 4.15. The *rightmost array* $\mathcal{R}(A)$ of an integer array A of length n over the alphabet $\{0, \ldots, n-1\}$ stores at position i the index of the rightmost occurrence of the integer i in A. If there is no occurrence of i in A then $A[i] = -1$. Formally stated:

$$\begin{aligned}
\mathcal{R}(A)[i] = k &\iff A[k] = i \ \wedge\ A[k'] \neq i\ \ \forall k' > k \\
\mathcal{R}(A)[i] = -1 &\iff A[k] \neq i\ \ \forall k
\end{aligned}$$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
A	2	3	2	10	14	6	7	15	7	15	11	9	1	0	2	0	1	9
$\mathcal{R}(A)$	15	16	14	1	-1	-1	5	8	-1	17	3	10	-1	-1	4	9	-1	-1

FIGURE 4.16: Example of rightmost array.

We now have defined all the necessary tools which will make it possible to define our algorithmic solution to the problems stated in Section 4.2.2.

4.3.2 Algorithms

Our algorithm to generate a function recognising abelian palindromic factors, relies on the construction of the prefix parity integer array $\hat{\mathbb{P}}_A(t)$. As shown in Lem. 4.13, we are

able to determine if $t[i\,..\,j]$ is abelian palindromic by evaluating the truthfulness of the expression $\hat{\mathbb{P}}_j(t) \oplus \hat{\mathbb{P}}_{i-1}(t) \in \{0\} \cup \{2^0, 2^1, \ldots, 2^{|\Sigma|-1}\}$.

Given $\hat{\mathbb{P}}_A(t)$, this expression may be evaluated in $\mathcal{O}(1)$ time for a given i and j. This follows from the fact that XOR is a constant time operation. Additionally, we may check if a positive integer is a power of 2 in constant time by employing a common bit manipulation tactic [75]. Namely, for a positive integer x, the expression $x \;\&\; (x-1)$ will evaluate to 0 if and only if x represents some integer power of 2, where & represents the bit-wise AND operation.

It is important to note, that these operations are constant time under the assumption that their arguments do not exceed the maximum word size w of the computer implementation used. If we assume that the alphabet size is bounded by the logarithm of n, then this assumption holds [76], i.e. $|\Sigma| \leq \log_2(n)$. Alternatively, the limitation on $|\Sigma|$ need not depend on n, and may instead be expressed in terms of the word size w of a machine. If the word size is w, we may assume these operations are constant for an alphabet size $|\Sigma| \leq w$. For a larger $|\Sigma|$, the expression in Lem. 4.13 may be evaluated in $\mathcal{O}(\frac{|\Sigma|}{w})$ time.

We now consider the construction of $\hat{\mathbb{P}}_A(t)$. It is possible to construct the array directly while maintaining a single instance of $\hat{\mathbb{P}}_i(t)$, by Lem. 4.17.

Lemma 4.17.

$$\hat{\mathbb{P}}_A(t)[0] = 2^{\text{ORD}(t[0])}$$
$$\hat{\mathbb{P}}_A(t)[i] = \hat{\mathbb{P}}_A(t)[i-1] + (2\,\mathbb{P}_i(t)[\text{ORD}(t[i])] - 1) \times 2^{\text{ORD}(t[i])} \quad 0 < i \leq n-1$$

Proof. The case for $\hat{\mathbb{P}}_A(t)[0]$ is trivially true. We note that $\hat{\mathbb{P}}_A(t)[i] = \hat{\mathbb{P}}_i(t)$ is an integer representation of $\mathbb{P}_i(t)$ interpreted as a binary string. $\mathbb{P}_i(t)$ and $\mathbb{P}_{i-1}(t)$ differ by a single bit flip, corresponding to the character encountered at $t[i]$. Therefore by Def. 4.8 and 4.9, $\hat{\mathbb{P}}_i(t)$ and $\hat{\mathbb{P}}_{i-1}(t)$ will accordingly differ by a single power of 2, specifically $2^{\text{ORD}(t[i])}$.

Whether $2^{\text{ORD}(t[i])}$ should be added or subtracted is dependant on the current parity of the character $t[i]$. This is determined by $\mathbb{P}_i(t)[\text{ORD}(t[i])]$, with 1 corresponding to addition $(+1)$ and 0 corresponding to subtraction (-1).

Thus the mapping $2b-1$ where $b \in \{0,1\}$ is the most recently flipped bit $b = \mathbb{P}_i(t)[\text{ORD}(t[i])]$, indicates the appropriate addition $(+1)$ or subtraction (-1). □

With this iterative equation for $\hat{\mathbb{P}}_A(t)$, we now have all the tools necessary to efficiently determine abelian palindromic factors and solve the problem as stated. We formalise the result in Theorem 4.18, addressing the first problem from Section 4.2.2.

Theorem 4.18. *Given a string t of length n over the alphabet Σ, after $\mathcal{O}(n)$ time preprocessing and $\mathcal{O}(n + |\Sigma|)$ space, we may perform queries to determine if $t[i\,..\,j]$ is abelian palindromic in $\mathcal{O}(1)$ time when $|\Sigma| \leq \log_2(n)$.*

Additionally with no constraint on the size of Σ, with $\mathcal{O}(\frac{|\Sigma|}{w}n)$ time preprocessing we may perform such queries in $\mathcal{O}(\frac{|\Sigma|}{w})$ time, where w is the computer word size.

Proof. By using Lem. 4.17 we may iteratively construct $\hat{\mathbb{P}}_A(t)[i]$ from $\hat{\mathbb{P}}_A(t)[i-1]$ in $\mathcal{O}(1)$ time at each step, while maintaining the Σ-sized data structure $\mathbb{P}_i(t)$, resulting in a total time complexity $\mathcal{O}(n)$ and space complexity $\mathcal{O}(n + |\Sigma|)$ to construct $\hat{\mathbb{P}}_A(t)$.

By evaluating the expression on $\hat{\mathbb{P}}_A(t)$ in Lem. 4.13, we may then determine if $t[i\,..\,j]$ is abelian palindromic. Evaluating this expression may be performed in $\mathcal{O}(1)$ time when the number of bits required to store $\hat{\mathbb{P}}_i(t)$ is no larger than a single computer word w, i.e. when $|\Sigma| \leq \log_2(n) \leq w$. In general, $\hat{\mathbb{P}}_i(t)$ may be stored in $|\Sigma|$ bits, requiring $\lceil \frac{|\Sigma|}{w} \rceil$ words to store, and thus a multiplying factor of $\mathcal{O}(\frac{|\Sigma|}{w})$ time is required for all operations involving $\hat{\mathbb{P}}_i(t)$, both when constructing $\hat{\mathbb{P}}_A(t)$ and when evaluating the expression in Lem. 4.13, corresponding to a single query. □

To address the second problem from Section 4.2.2, we consider an algorithm to generate the abelian palindromic array which makes use of Theorem 4.18 and the rightmost array described in Def. 4.15. We also make use of Lem. 4.19.

Lemma 4.19. *The abelian palindromic array A of a string t satisfies:*

$$A[i] = \max\{\mathcal{R}(j) : j \in M(t, \hat{\mathbb{P}}_A(t)[i])\}$$

Where M is the match set as described in Lem. 4.14.

Proof. Lem. 4.14 indicates that the longest abelian palindromic factor occurring at i is $t[i\,..\,j]$ where j is the index of the rightmost prefix parity integer with a value contained in the match set $M(t, i)$.

By Def. 4.15, this rightmost j can be found by taking the largest value obtained when querying the rightmost array with every member of the match set. □

Theorem 4.20. *Given a string t of length n over the alphabet Σ, we may determine the abelian palindromic array of t in $\mathcal{O}(|\Sigma|n)$ time and $\mathcal{O}(n+|\Sigma|)$ space, when $|\Sigma| \leq \log_2(n)$.*

Proof. Via the proof in Theorem 4.18 we are able to calculate the prefix parity integer array $\hat{\mathbb{P}}_A(t)$ in $\mathcal{O}(n)$ time and with $\mathcal{O}(n + |\Sigma|)$ space.

Since $|\Sigma| \leq \log_2(n)$, we know all values of $\mathcal{R}(A)[i] \in \{-1, 0, \ldots, n-1\}$. Therefore the rightmost array $\mathcal{R}(\hat{\mathbb{P}}_A(t))$ may be calculated in $\mathcal{O}(n)$ time, by parsing $\hat{\mathbb{P}}_A(t)$ from right to left and storing any new values encountered. Full details are available in the pseudo-code in Section 4.4.

We now apply Lem. 4.19, which enables us to determine the longest abelian palindromic factor occurring at i by performing $|\Sigma|$ constant time queries. Thus a total of $\mathcal{O}(|\Sigma|n)$ constant time queries are required, and the total time complexity to generate the abelian palindromic array is $\mathcal{O}(|\Sigma|n)$. □

4.4 Pseudo-code

The pseudo-code for both of the problems presented in Section 4.2.2 are shown here. The first problem of Abelian Palindromic Factor Recognition may be addressed using Theorem 4.18, through the use of the single function getPrefixParityIntegerArray.

The second problem of generating the Abelian Palindromic Array is addressed using the Theorem 4.20, through the use of the function getAbelianPalindromicArray, which requires an additional two subroutines getRightMostArray and getMatchSet, which are also defined within the pseudo-code.

Function GETPREFIXPARITYINTEGERARRAY(t, Σ)

input :

t = Text to be processed.
Σ = Alphabet over which text t is defined.

output:

The prefix parity integer array of the text t.

$n \leftarrow |t|$
$\sigma \leftarrow |\Sigma|$
$A \leftarrow$ integer array of length n filled with 0
$B \leftarrow$ integer array of length n filled with 0
$B[0] = 1$

for $i \leftarrow 1$ **to** $\sigma - 1$ **do**
    $B[i] = 2 \times B[i-1]$

$\mathbb{P} \leftarrow$ boolean array of length σ filled with 0
prev $\leftarrow 0$

for $i \leftarrow 0$ **to** $n - 1$ **do**
    if $\mathbb{P}[\text{ORD}(t[i])] == 1$ **then**
        $A[i] = \text{prev} - B[\text{ORD}(t[i])]$
    else
        $A[i] = \text{prev} + B[\text{ORD}(t[i])]$
    $\mathbb{P}[\text{ORD}(t[i])] =$ **not** $\mathbb{P}[\text{ORD}(t[i])]$
    prev $= A[i]$

return A

Function GETRIGHTMOSTARRAY(A)

input :

A = Array of integers.

output:

The rightmost array of the array A.

$n \leftarrow |A|$
$R \leftarrow$ integer array of length n filled with -1

for $i \leftarrow n-1$ **to** 0 **do**
  if $R[A[i]] == -1$ **then**
    $R[A[i]] = i$

return R

Function GETMATCHSET(x, n)

input :

x = Integer for which match set is calculated.
n = Size of final match set.

output:

The match set of x.

$M \leftarrow$ integer array of length $n+1$ filled with 0

for $i \leftarrow 0$ **to** $n-1$ **do**
  $M[i] = x \oplus 2^i$

$M[n] = x$

return M

Function GETABELIANPALINDROMICARRAY(t, Σ)

input :

t = Text to be processed.
Σ = Alphabet over which text t is defined.

output:

The prefix parity integer array of the text t.

$n \leftarrow |t|$
$\sigma \leftarrow |\Sigma|$
$A \leftarrow$ GETPREFIXPARITYINTEGERARRAY(t, Σ)
$R \leftarrow$ GETRIGHTMOSTARRAY(A)
$P \leftarrow$ integer array of length n filled with 0

for $i \leftarrow 0$ **to** $n - 1$ **do**
    $M \leftarrow$ GETMATCHSET$(A[i-1], \sigma)$
    rightmost $\leftarrow -1$
    for each match **in** M **do**
        if R[match] $>$ rightmost **then**
            rightmost $= R$[match]
    if rightmost $> i - 1$ **then**
        $P[i] =$ rightmost $- i + 1$
    else
        $P[i] = 0$

return P

4.5 Conclusion

We have presented a novel problem, first by defining abelian palindromes as a new type of data structure, and then by considering methods of recognising such strings. In order to carry out this task, we made use of the Parikh vector, in addition to defining several new yet related vector structures, along with a suggested set of notations.

This work presents an initial exploration of the mathematical properties of such strings, and a theoretically efficient method of identifying them. Specifically, two algorithms have been presented, the first for recognising whether or not a factor is abelian palindromic, and the second for generating an array which provides the length of the longest abelian palindromic factor at each position in a string.

The proposed algorithms are both dependant on a new data structure called the prefix parity integer array, requiring $\mathcal{O}(n)$ time to compute for a string with an alphabet size $|\Sigma| \leq \log_2(n)$. Additional complexity is required to determine the longest abelian palindromic factor for each position, namely $\mathcal{O}(|\Sigma|n)$ time.

Though the pseudocode has been presented for both algorithms, in depth performance testing has not been carried out, and may represent an opportunity for further research. However, a basic implementation was created in order to verify the correctness of the algorithm, in addition to the stated proof in 4.18.

The potential applications for abelian palindromes are as yet unknown, though we hope this work acts as a springboard for potentially related problems in the future. Additionally, it may prove useful to consider this work in conjunction with the study of other data structures with abelian properties.

The main improvement in this work, would be to remove the need for the current requirement that $|\Sigma| \leq \log_2(n)$, in order to obtain our current best complexity time, which appears to be a reasonable goal. Some additional performance testing on relevant data might also serve to determine the practical use of such algorithms. Finally, it would certainly prove interesting to consider an alternative solution in light of the development of quantum computing algorithms. Both algorithms presented here are dependant on the properties of bit manipulation within the classical computing model. It remains to be seen if the use of qubits could provide an alternative or more efficient solution [77, 78].

Chapter 5

Maximal Palindromes

A maximal palindromic factor is a factor (or substring) which reads the same left to right and can be extended no further in either direction [79]. In this chapter, the focus will be on creating an algorithm that is capable of producing a sequence of maximal palindromic factors that corresponds to a given degenerate string, i.e. a string which may have some positions with multiple possible characters. An attempt is additionally made to minimise the number of these factors. As part of this process, a complete list of all maximal palindromes in the string may also be generated. Specifically, this chapter presents an $\mathcal{O}(k|\Sigma|(k + \log|\Sigma|) + kn)$ time and $\mathcal{O}(k(k + |\Sigma|) + n)$ space algorithm to achieve this, along with the pseudocode of an implementation.

5.1 Background

Finding string regularities is essential in many applications such as detecting text errors, musical analysis and molecular biology [80]. As a result, there are many scientific fields devoted to the study of changes in pattern sequences and their effect. For example, computational biology depends on analytical methods and developing algorithms to study regularities in DNA sequences like repeats and covers, particularly within degenerate strings, which assist in the discovery of genetic characteristics [81, 82].

Within the literature, the term *degenerate* string or *indeterminate* string is used to refer to a generalised string in which individual characters may have multiple values [83–85]. We may consider such strings as a general model for strings in which multiple nucleotides are represented with a single ambiguity character.

Many studies over the last decade have drawn attention to the importance of finding efficient algorithms that deal with degenerate strings. Notably, there is a need to use

degenerate strings for various applications such as search engines, cryptoanalysis [83] and particularly genetic science.

From a biological perspective, the study of palindromes has become a necessary development, for instance where palindromes act as an indicator of the probability of cancer existence [72]. Additionally, genetic palindromes may correspond to the occurrence of endonuclease restriction sites, caused by restriction enzymes. For example, we see a statistical analysis of the complete genome of Escherichia coli, in which palindromic sequences occur frequently within regions of interest [86].

Furthermore, Kolpakov and Kucherov in [87] proposed two algorithms for computing two types of gapped palindromes which are called *long-armed palindromes* and *length constrained palindromes*. These algorithms are important in the analysis of DNA and RNA sequences that have palindromic sequences of nucleotides and a gap between the left and right copies of palindromes.

Alatabbi et. al in [79] presented an algorithm for finding the maximal palindromic factorisation of a finite string in linear time. The contribution of this chapter is to present a more general method for decomposing a degenerate string into a sequence of maximal palindromic factors.

5.2 Problem Outline

5.2.1 Terminology

As eluded to in the preamble, we may make use of *degenerate strings* to model strings with variable possible characters at some positions. A *degenerate symbol* $\tilde{\sigma}$ over an alphabet Σ is a non-empty subset of Σ, i.e. $\tilde{\sigma} \subseteq \Sigma$ and $\tilde{\sigma} \neq \emptyset$. $|\tilde{\sigma}|$ denotes the size of the set and we have $1 \leq |\tilde{\sigma}| \leq |\Sigma|$. A finite sequence $\tilde{x} = \tilde{x}_0\tilde{x}_1 \dots \tilde{x}_{n-1}$ is said to be a *degenerate string* (also known as an *indeterminate string*) if $\tilde{x}_i$ is a degenerate symbol for each $0 \leq i \leq n-1$. A degenerate string is built over the potential $2^{|\Sigma|} - 1$ non-empty subsets of characters belonging to Σ. The *length* of a degenerate string $\tilde{x}$ is the number of degenerate symbols n.

For example, $\tilde{x} = [\mathtt{A}, \mathtt{C}]\,[\mathtt{A}]\,[\mathtt{G}]\,[\mathtt{C}, \mathtt{C}]\,[\mathtt{A}]\,[\mathtt{A}, \mathtt{C}, \mathtt{G}]$ is a degenerate string of length 6 over the alphabet $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}\}$ (or $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ with no occurrences of $\mathtt{T}$). If $|\tilde{x}_i| = 1$, that is $\tilde{x}$ represents a single character of Σ, we say that $\tilde{x}_i$ is a *solid symbol* and i is a *solid position*. Otherwise $\tilde{x}_i$ and i are said to be a *non-solid symbol* and *non-solid position* respectively. For convenience we often write $\tilde{x}_i = \sigma$ $(\sigma \in \Sigma)$, instead of $\tilde{x}_i = [\sigma]$, in the case of solid symbols. Consequently, the previous example $\tilde{x}$ may be written as

$\tilde{x} = [\mathtt{A}, \mathtt{C}]\,\mathtt{A}\,\mathtt{G}\,[\mathtt{C}, \mathtt{C}]\,\mathtt{A}\,[\mathtt{A}, \mathtt{C}, \mathtt{G}]$. A degenerate string containing only solid symbols is a *solid string* and behaves the same as a classical string of characters, and for such strings we may omit the $\sim$ notation. In addition, a solid symbol $[\sigma]$ and its corresponding character $\sigma \in \Sigma$ may be treated as interchangeable for our purposes.

A degenerate string $\tilde{x}$ is called *conservative* if the number of non-solid symbols in $\tilde{x}$ is known to be no more than some natural number k. The concatenation of degenerate strings $\tilde{x}$ and $\tilde{y}$ is $\tilde{x}\tilde{y}$. A degenerate string $\tilde{v}$ is a substring of a degenerate string $\tilde{x}$ if $\tilde{x} = \tilde{u}\tilde{v}\tilde{w}$ for some degenerate strings $\tilde{u}$ and $\tilde{w}$. By $\tilde{x}[i\,..\,j]$ we represent a substring $\tilde{x}_i\tilde{x}_{i+1}\ldots\tilde{x}_j$ of $\tilde{x}$.

For degenerate strings, the notion of character equality is extended to symbol equality between two degenerate symbols. Two degenerate symbols $\tilde{x}$ and $\tilde{y}$ are said to *match* (denoted $\tilde{x} \approx \tilde{y}$) if $\tilde{x} \cap \tilde{y} \neq \emptyset$. Extending this notion to degenerate strings, we say that two degenerate strings $\tilde{x}$ and $\tilde{y}$ match (denoted $\tilde{x} \approx \tilde{y}$) if $|\tilde{x}| = |\tilde{y}|$ and all corresponding symbols in $\tilde{x}$ and $\tilde{y}$ match. Note that the relation $\approx$ is not transitive. A degenerate string $\tilde{x}$ is said to *occur* at position i in another degenerate string $\tilde{y}$ if $\tilde{x} \approx \tilde{y}[i\,..\,i+|\tilde{x}|-1]$.

We demonstrate an example of degenerate string notation in Fig. 5.1, where a vector of characters is used to indicate the possible values of a non-solid symbol.

$$\begin{pmatrix}\mathtt{A}\\ \mathtt{C}\end{pmatrix}\mathtt{AG}\begin{pmatrix}\mathtt{C}\\ \mathtt{G}\end{pmatrix}\mathtt{A}\begin{pmatrix}\mathtt{A}\\ \mathtt{C}\\ \mathtt{G}\end{pmatrix}$$

FIGURE 5.1: Degenerate string of length $n = 6$ over $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ with $k = 3$.

A *palindromic factor* of a degenerate string $\tilde{x}$ is some solid string $p \approx \tilde{x}[i\,..\,j]$ such that p is equal to its reversal ($p = p^R$). Equivalently we can define an even length palindromic factor p as a factor that can be expressed in the form ww^R for some string w, and an odd length palindromic factor p as a factor that can be expressed in the form wcw^R for some string w and a single character c. The *centre* of a palindromic factor $p \approx \tilde{X}[i\,..\,j]$ is defined as $\frac{i+j}{2}$, and its *radius* is defined as $\frac{|p|}{2}$.

A *maximal palindromic factor* of a degenerate string $\tilde{x}$ is the longest palindromic factor at a given centre, which can be extended no further. In other words $p \approx \tilde{x}[i\,..\,j]$ is maximal for its centre $\frac{i+j}{2}$ if $\tilde{x}[i-1] \not\approx \tilde{x}[j+1]$ or if either $i-1$ or $j+1$ are invalid indexes of $\tilde{x}$.

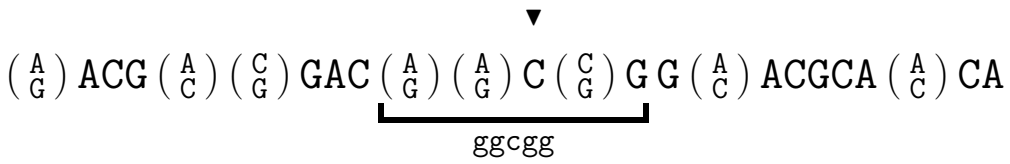

FIGURE 5.2: Maximal palindrome for centre 11.

For example in Fig. 5.2 we show a palindromic factor at centre 11 corresponding to $\tilde{x}[9\,..\,13]$. In fact this is a maximal palindromic factor, as it can not be extended further within $\tilde{x}$ while still containing a palindrome, i.e. $\tilde{x}[8] = \mathtt{C} \not\approx \mathtt{G} = \tilde{x}[14]$.

We denote by $\mathrm{MP}(\tilde{x})$ the set of all maximal palindromes of $\tilde{x}$, i.e. the set containing all pairs (i, j) such that $\tilde{x}[i\,..\,j]$ is a maximal palindrome in $\tilde{x}$. By considering the number of valid centres, it is clear that the number of unique maximal palindromic factors must be no more than $2n - 1$, i.e. $|\mathrm{MP}(\tilde{x})| \leq 2n - 1$.

Palindromic factorisation is the process of splitting a string into adjacent factors such that all factors are palindromes and the minimal number of factors possible is used. We may add the further requirement that all palindromes in the factorisation are maximal palindromes [88]. This problem can be extended to degenerate strings, as in Def. 5.3.

Definition 5.3. Given a degenerate string $\tilde{x}$, we define a *maximal palindromic factorisation* as a sequence of m factors $f_1, f_2, \ldots, f_m$ of $\tilde{x}$ such that:

1. $\tilde{x} = f_1\, f_2\, \ldots\, f_m$
2. $f_i \in \mathrm{MP}(\tilde{x})\ \forall i \in \{1\,..\,m\}$
3. m is minimised

Note that for Def. 5.3, there is a possibility of multiple valid factorisations of a string $\tilde{x}$ with the same minimal m. Alternatively there may be no valid maximal palindromic factorisation of $\tilde{x}$, which differs to ordinary palindromic factorisation. This is a result of the fact that a single character is guaranteed to be an ordinary palindrome, but not necessarily a maximal palindrome.

We show an example of a maximal palindromic decomposition in Fig. 5.4. Here the minimal possible number of palindromic factors (6) is obtained.

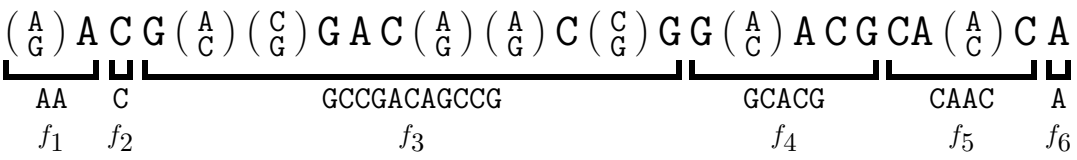

FIGURE 5.4: Maximal palindromic decomposition.

5.2.2 Problem Statement

We now formally define the problem addressed within this chapter, namely Degenerate Maximal Palindromic Factorisation:

DEGENERATE MAXIMAL PALINDROMIC FACTORISATION

Input:
Degenerate string $\tilde{x}$ of length n and a natural number k representing an upper bound on the number of non-solid symbols in $\tilde{x}$.

Output:
Array of indexes $p_0, p_1, \ldots, p_m$ describing a maximal palindromic factorisation $f_1, f_2, \ldots, f_m$ where $f_i = \tilde{x}[p_{i-1} \,..\, p_i - 1]$ is a maximal palindrome for all $i \in [1, m]$ and the number of factors m is minimised.

As an example of a factorisation $f_1, f_2, \ldots, f_m$, for the factorisation of the degenerate string described in Fig. 5.4, the corresponding output is the array $[0, 2, 3, 14, 19, 23, 24]$.

5.3 Solution

5.3.1 Tools

We first preprocess the string $\tilde{x}$, to obtain a string we call the *solid equivalent* of $\tilde{x}$, denoted $x_{\$}$. To obtain $x_{\$}$ we transform $\tilde{x}$ into a solid string by replacing each of its k non-solid symbols with a unique character $\$_d$, where $0 \leq d < k$. These new characters $\$_0, ..., \$_{k-1}$ are defined to not match each other and not match any characters of the original alphabet Σ of $\tilde{x}$. The location in $x_{\$}$ of a given $\$_d$ is denoted by $\texttt{loc}(\$_d)$.

$$x_{\$}[i] = \begin{cases} \tilde{x}[i] & \text{for } \tilde{x}[i] \text{ solid} \\ \$_d & \text{for } \tilde{x}[i] \text{ non-solid, where } d \text{ non-solid symbols occur left of } \tilde{x}[i] \end{cases}$$

Where $\$_0, ..., \$_{k-1}$ are defined such that $\$_i = \$_j \implies i = j$ and $\$_d \notin \Sigma \ \ \forall d$.

$$\texttt{loc}(\$_d) \text{ is defined such that } x_{\$}[\texttt{loc}(\$_d)] = \$_d$$

Next we create a new string $s = x_{\$}\, \#_1\, {x_{\$}}^R\, \#_2$, where $\#_1$ and $\#_2$ are new characters that match no others in $\Sigma \cup \{\$_0, ..., \$_{k-1}\}$ and $\#_1 \neq \#_2$. We then preprocess s and construct its suffix tree, so that we may perform lce queries on pairs of substrings of s, also known as longest common extensions, as defined by Def. 2.7.

We define the lexicographical ordering of the alphabet to be $\#_1, \#_2, \$_0, \ldots, \$_{k-1}, \sigma_0, \ldots, \sigma_{|\Sigma|-1}$, where σ_i refers to the ith character in the lexicographical ordering of the alphabet Σ (0-indexed).

In the process of constructing $x_{\$}$ and s, we also build a matching table $\mathcal{M}$ over the alphabet $\Sigma \cup \{\$_0, ..., \$_{k-1}\} \cup \{\#_1, \#_2\}$. We define the matching table as follows:

$$\mathcal{M} : \Sigma \cup \{\$_0, ..., \$_{k-1}\} \cup \{\#_1, \#_2\} \times \Sigma \cup \{\$_0, ..., \$_{k-1}\} \cup \{\#_1, \#_2\} \to \{\texttt{true}, \texttt{false}\}$$

$$\mathcal{M}(s_1, s_2) = \begin{cases} s_1 = s_2 & \text{for } s_1, s_2 \in \Sigma \\ \texttt{loc}(s_1) = \texttt{loc}(s_2) & \text{for } s_1, s_2 \in \{\$_0, ..., \$_{k-1}\} \\ \tilde{x}[\texttt{loc}(s_1)] \approx s_2 & \text{for } s_1 \in \{\$_0, ..., \$_{k-1}\}, s_2 \in \Sigma \\ \tilde{x}[\texttt{loc}(s_2)] \approx s_1 & \text{for } s_1 \in \Sigma, s_2 \in \{\$_0, ..., \$_{k-1}\} \\ s_1 = s_2 & \text{for } s_1 \in \{\#_1, \#_2\} \vee s_2 \in \{\#_1, \#_2\} \end{cases}$$

Note that it trivially follows from the commutativity of operations in the definition, that $\mathcal{M}(s_1, s_2) = \mathcal{M}(s_2, s_1)$ for any pair of characters $s_1, s_2 \in \Sigma \cup \{\$_0, ..., \$_{k-1}\} \cup \{\#_1, \#_2\}$. We say that two strings x and y match over the matching table $\mathcal{M}$ denoted $x =_{\mathcal{M}} y$ if they match at all positions over $\mathcal{M}$.

This matching table $\mathcal{M}$ will be used to extend the definition of lce as defined in Def. 2.7, by incorporating $\mathcal{M}$ as an optional additional parameter of the lce function, such that the matching condition of two prefixes within the string s generated from $\tilde{x}$ is determined by $\mathcal{M}$.

Definition 5.5. For a given degenerate string $\tilde{x}$ of length n and two indexes and some matching table $\mathcal{M}$ over the alphabet of x, we define the longest common extension $\text{lce}(s, i, j, \mathcal{M})$ as follows:

$$\text{lce}(s, i, j, \mathcal{M}) = \max(\{l : s[i\,..\,i+l-1] =_{\mathcal{M}} s[j\,..\,j+l-1]\} \cup \{0\})$$

$$s = x_{\$}\,\#_1\,x_{\$}{}^R\,\#_2$$

5.3.2 Algorithm

Following from Section 5.3.1, we now have $x_{\$}$, s, $\mathcal{M}$ and the ability to perform lce operations on s in constant time using the matching table. This gives us the necessary information to systematically find the maximal palindrome at each centre of $\tilde{x}$. This is done via repeated lce queries on s. If we choose our lce queries strategically, we may determine the radius of maximal palindromes of $\tilde{x}$ at each centre.

Lemma 5.6. *Given a degenerate string $\tilde{x}$ of length n with k non-solid symbols, the maximal palindrome p at the centre $c \in \{0, \frac{1}{2}, 1, \frac{3}{2} \ldots, n-1\}$ of $\tilde{x}$ is $p = \tilde{x}[i\,..\,j]$ if $i \leq j$ or $p = \epsilon$ if $i > j$ where i and j satisfy the following:*

$$i = \lceil c \rceil - e$$
$$j = \lfloor c \rfloor + e$$

$$e = \text{lce}(s, \lceil c \rceil, 2n - \lfloor c \rfloor, \mathcal{M})$$

$$s = x_{\$}\,\#_1\,x_{\$}^R\,\#_2$$

Proof. We split the problem into two separate cases for even length and odd length palindromes.

For an even length palindrome, the centre $c = c' + \frac{1}{2}$ for some $c' \in \{0, \ldots, n-2\}$. We calculate e, the length of the longest pair of matching factors within x starting from index $c + \frac{1}{2}$ reading to the right and from index $c - \frac{1}{2}$ reading to the left. Since $s = x_{\$}\,\#_1\,x_{\$}^R\,\#_2$ is of length $2n + 2$, we express this as $e = \text{lce}(s, c + \frac{1}{2}, 2n - c + \frac{1}{2}, \mathcal{M})$. The maximal palindrome at centre c then corresponds to $p = \tilde{x}[c + \frac{1}{2} - e\,..\,c - \frac{1}{2} + e]$ if $e > 0$ or $p = \epsilon$ if $e = 0$.

For an odd length palindrome, the centre $c \in \{0, \ldots, n-1\}$. We calculate e, the length of the longest pair of matching factors within x starting from index c reading to the right

and from index c reading to the left. Since $s = x_{\$}\,\#_1\,x_{\$}^R\,\#_2$ is of length $2n+2$, we express this as $e = \text{lce}(s, c, 2n-c, \mathcal{M})$. The maximal palindrome at centre c then corresponds to $p = \tilde{x}[c-e\,..\,c+e]$.

We may combine both the results for even and odd length palindromes into the statement of the lemma, through use of ceiling and floor notation. □

Thus we have a systematic way to determine the $2n-1$ maximal palindromes $\text{MP}(\tilde{x})$ of any degenerate string $\tilde{x}$ of length n. An example of the maximal palindromes of a degenerate string may be seen in Fig. 5.8.

Having computed all maximal palindromes $\text{MP}(\tilde{X})$, we now construct a graph, where indexes of the string $\tilde{x}$ correspond to vertices and maximal palindromes correspond to edges. Specifically we build a graph G with vertices V and edges E, where $V = \{0, \ldots, n\}$ and $E = \{(i, j+1) : (i, j) \in \text{MP}(\tilde{x})\}$.

Once the graph G is constructed, a breath first search is performed to find the shortest path from the source vertex 0 to the sink vertex n [79]. The list of vertices encountered on the shortest path in order of occurrence, are labelled $P_0, P_1, \ldots, P_m$. These correspond to a list of m factors $f_1, f_2, \ldots, f_m$ constituting the maximal palindromic factorisation where $f_i = \tilde{x}[P_{i-1}\,..\,P_i - 1]$.

Note that if there are multiple possible factorisations with the same minimal number of factors m, the specific factorisation returned will depend on the implementation of the breadth first search. The algorithm may potentially be modified to return all possible factorisations with the minimal number of factors. Additionally, if no valid path exists between vertices 0 and n due to the graph being disconnected, then no valid factorisation exists.

Theorem 5.7. *Given a degenerate string $\tilde{x}$ of length n over the alphabet Σ and a natural number k representing an upper bound on the number of non-solid symbols in $\tilde{x}$, the maximal palindromic factorisation of $\tilde{x}$ may be found in $\mathcal{O}(k|\Sigma|(k + \log|\Sigma|) + kn)$ time and $\mathcal{O}(k(k + |\Sigma|) + n)$ space.*

Proof. We analyse the various steps of the algorithm as set out in Section 5.3.2. To obtain the solid equivalent $x_{\$}$ of $\tilde{x}$, construct the string $s = x_{\$}\,\#_1\,x_{\$}^R\,\#_2$ and construct the suffix tree of x, each require $\mathcal{O}(n)$ time and space.

It is reasonable to assume that non-solid symbols store their characters in lexicographical order. Under this assumption, determining a match between any two non-solid symbols is an $\mathcal{O}(|\Sigma|)$ operation, and between a non-solid and solid symbol is an $\mathcal{O}(\log(|\Sigma|))$ operation. It therefore follows that the preprocessing time to generate the portion of

the matching table $\mathcal{M}$ comparing 2 symbols where at least 1 is a non-solid symbol is $\mathcal{O}(k|\Sigma|(k + \log|\Sigma|))$. Since the portion of the matching table comparing characters $s_1, s_2 \in \Sigma$ is already known, no further preprocessing is required. Indeed, the required additional storage space to implicitly store the matching table $\mathcal{M}$ is thus $\mathcal{O}(k(k+|\Sigma|))$, since we require $\mathcal{O}(k^2)$ space for storing all match results for pairs of non-solid symbols and $\mathcal{O}(k|\Sigma|)$ to store all match results for pairs of symbols consisting of a solid and a non-solid symbol.

Determining maximal palindromes requires us to consider each of the $2n-1$ possible centres of $\tilde{x}$. For each of these centres, we perform a single appropriate $\text{lce}(s, i, j, \mathcal{M})$ query on the string $s = x_{\$}\,\#_1\,{x_{\$}}^R\,\#_2$. This is done by calculating $\text{lce}_{k'}(s, i, j)$ for $k' \in [0, k]$, and observing the following:

$$\text{lce}(s, i, j, \mathcal{M}) = \max(\{l_{k'} : \mathcal{M}(i + l_{k'-1} + 1, j + l_{k'-1} + 1) = \texttt{true}, k' \in [1, k]\} \cup \{l_0\})$$

$$l_{k'} = \text{lce}_{k'}(s, i, j)$$

Thus a single $\text{lce}(s, i, j, \mathcal{M})$ query requires $\mathcal{O}(k)$ time. Therefore we may determine the list of maximal palindromes $\text{MP}(\tilde{x})$ in $\mathcal{O}(kn)$ time and $\mathcal{O}(n)$ space, after initial preprocessing.

The construction of the graph G requires $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ space. Performing a single breadth first search on G is a standard $\mathcal{O}(n)$ time operation since the number of edges $|E|$ and number of vertices $|V|$ are bounded by $\mathcal{O}(n)$.

Thus by considering the total complexity of these individual steps, the final complexity of the algorithm follows.

□

5.4 Pseudo-code

The entry point of the pseudocode is the function GETFACTORISATION. This accepts an array $\tilde{x}$ of length n where every element in the array is a string representing the possible characters of a degenerate symbol. A string of length 1 is thus a solid symbol. The input parameter k specifies an upper bound on the number of non-solid symbols in $\tilde{x}$. An array of integers defining the optimal factorisation is returned. Note that if the string $\tilde{x}$ does not have any possible factorisation into only maximal palindromes, the function returns `null`. We make use of some functions for which the pseudocode is not given. We detail those functions here:

LCE(x, i, j)

Function to calculate longest common extension. Given a solid string x returns the length of the longest common prefix between the ith and jth suffix of x. Open source implementations are available using suffix trees or suffix arrays.

getMatchTable($\tilde{x}, n, k$)

Given a degenerate string $\tilde{x}$ of length n over the alphabet Σ and an upper bound on non-solid symbols k, returns the matching table $\mathcal{M}$ over the alphabet $\Sigma \cup \{\$_0, ..., \$_{k-1}\} \cup \{\#_1, \#_2\}$. The specific implementation of this function and the returned matching table data structure is dependent on the data structures used to implement degenerate strings. Given any two symbols s_1 and s_2 from the set $\Sigma \cup \{\$_0, ..., \$_{k-1}\} \cup \{\#_1, \#_2\}$, $\mathcal{M}(s_1, s_2)$ gives a `true` or `false` value indicating whether or not s_1 matches s_2.

getShortestPath(G, *source*, *target*)

Given a directed graph G of vertices and edges, returns an array of integers containing the sequence of vertex labels encountered on the shortest path from the *source* vertex to the *target* vertex. If no possible path exists it returns `null`. If multiple shortest paths exist, the chosen path will depend on the specifics of the implementation. For example the array $[0, 2, 7, 10]$ is returned if the shortest path from *source* vertex 0 to *target* vertex 10 is via vertices 2 and 7. The shortest path can be found by performing a breadth first search on G starting at *source* and choosing the path that first encounters the *target* during the search. Open source implementations of breath first search are available.

Function GETFACTORISATION($\tilde{x}$, n, k)

input :

$\tilde{x}$ = Degenerate string to be searched.

n = Length of the degenerate string $\tilde{x}$.

k = Upper bound on non-solid symbols.

output:

An array of integers representing the maximal palindromic factorisation of $\tilde{x}$.

$x_\$ \leftarrow$ empty string

$j \leftarrow 0$

for $i \leftarrow 0$ **to** $n-1$ **do**
    char $\leftarrow \tilde{x}[i]$
    if LENGTH(char) == 1 **then**
        $x_\$ = x_\$ +$ char
    else
        $x_\$ = x_\$ + \$_j$
        $j = j + 1$

$\mathcal{M} \leftarrow$ GETMATCHTABLE$(\tilde{x}, n, k)$

$s \leftarrow x_\$ + \#_1 +$ REVERSE$(x_\$) + \#_2$

even_palindromes $\leftarrow$ GETPALINDROMES$(s, n, k, 0, \mathcal{M})$

odd_palindromes $\leftarrow$ GETPALINDROMES$(s, n, k, 1, \mathcal{M})$

$G \leftarrow$ directed graph with vertices 0 to n

foreach p **in** even_palindromes $\cup$ odd_palindromes **do**
    if p **is not** NULL **then**
        add edge $(p.left, p.right + 1)$ to G

return GETSHORTESTPATH$(G, 0, n)$

Function GETPALINDROMES(s, n, k, isOdd, $\mathcal{M}$)

input :

s = Text generated from a degenerate string.

n = Length of the original degenerate string.

k = Upper bound on non-solid symbols.

isOdd = Boolean value specifying request for odd or even palindromes.

$\mathcal{M}$ = Matching table.

output:

A set of pairs (i, j) representing the list of maximal palindromes found, where i and j are the start and end index of the palindrome.

palindromes $\leftarrow$ array of length $(n - 1 + \text{isOdd})$ of pairs $(0, 0)$

for $i \leftarrow 1$ **to** $n - 1 + \text{isOdd}$ **do**

$j \leftarrow 2n - i + 1 + \text{isOdd}$

$e \leftarrow$ REALLCE$(s, 2n + 2, k, i, j, \mathcal{M})$

left $\leftarrow i - e - \text{isOdd}$

right $\leftarrow i + e - 1$

if left $\leq$ right **then**

palindromes$[i - 1]$ = (left, right)

else

palindromes$[i - 1]$ = NULL

return *palindromes*

Function REALLCE(s, n, k, i, j, $\mathcal{M}$)

input :

s = Text generated from a degenerate string.
n = Length of the original degenerate string.
k = Upper bound on non-solid symbols.
i = Index within the original degenerate string.
j = Index within the original degenerate string.
$\mathcal{M}$ = Matching table.

output:

The longest common extension at i and j within the original degenerate string.

real_lce $\leftarrow 0$

mismatch_count $\leftarrow 0$

while mismatch_count $< k + 1$ **do**
    real_lce $\leftarrow$ real_lce + LCE$(s, i + \text{real_lce}, j + \text{real_lce})$
    if $i + \text{real\_lce} \geq n$ *or* $j + \text{real\_lce} \geq n$ **then**
        break
    $s_1 \leftarrow s[i + \text{real\_lce}]$ $s_2 \leftarrow s[j + \text{real\_lce}]$
    if $\mathcal{M}(s_1, s_2)$ **then**
        real_lce = real_lce +1
    else
        break
    mismatch_count = mismatch_count + 1
return real_lce

Centre	Maximal Palindrome	Edge		
0.0	A	0	⟶	1
0.5	AA	0	⟶	2
1.0	A	1	⟶	2
1.5	ϵ		–	
2.0	C	2	⟶	3
2.5	ϵ		–	
3.0	CGC	2	⟶	5
3.5	ϵ		–	
4.0	GAG	3	⟶	6
4.5	GCCG	3	⟶	7
5.0	C	5	⟶	6
5.5	AGGA	4	⟶	8
6.0	G	6	⟶	7
6.5	ϵ		–	
7.0	A	7	⟶	8
7.5	ϵ		–	
8.0	GCCGACAGCCG	3	⟶	14
8.5	ϵ		–	
9.0	A	9	⟶	10
9.5	CAAC	8	⟶	12
10.0	A	10	⟶	11
10.5	ϵ		–	
11.0	GGCGG	9	⟶	14
11.5	ACGGCCGGCA	7	⟶	17
12.0	C	12	⟶	13
12.5	GG	12	⟶	14
13.0	ACGGGCA	10	⟶	17
13.5	CGGC	12	⟶	16
14.0	G	14	⟶	15
14.5	ϵ		–	
15.0	A	15	⟶	16
15.5	AA	15	⟶	17
16.0	GCACG	14	⟶	19
16.5	ϵ		–	
17.0	C	17	⟶	18
17.5	ϵ		–	
18.0	AACGCAA	15	⟶	22
18.5	ϵ		–	
19.0	C	19	⟶	20
19.5	ϵ		–	
20.0	CAC	19	⟶	22
20.5	CAAC	19	⟶	23
21.0	A	21	⟶	22
21.5	ACCA	20	⟶	24
22.0	ACA	21	⟶	24
22.5	ϵ		–	
23.0	A	23	⟶	24

FIGURE 5.8: Table of maximal palindromes and associated graph edges for the example string from Fig. 5.2.

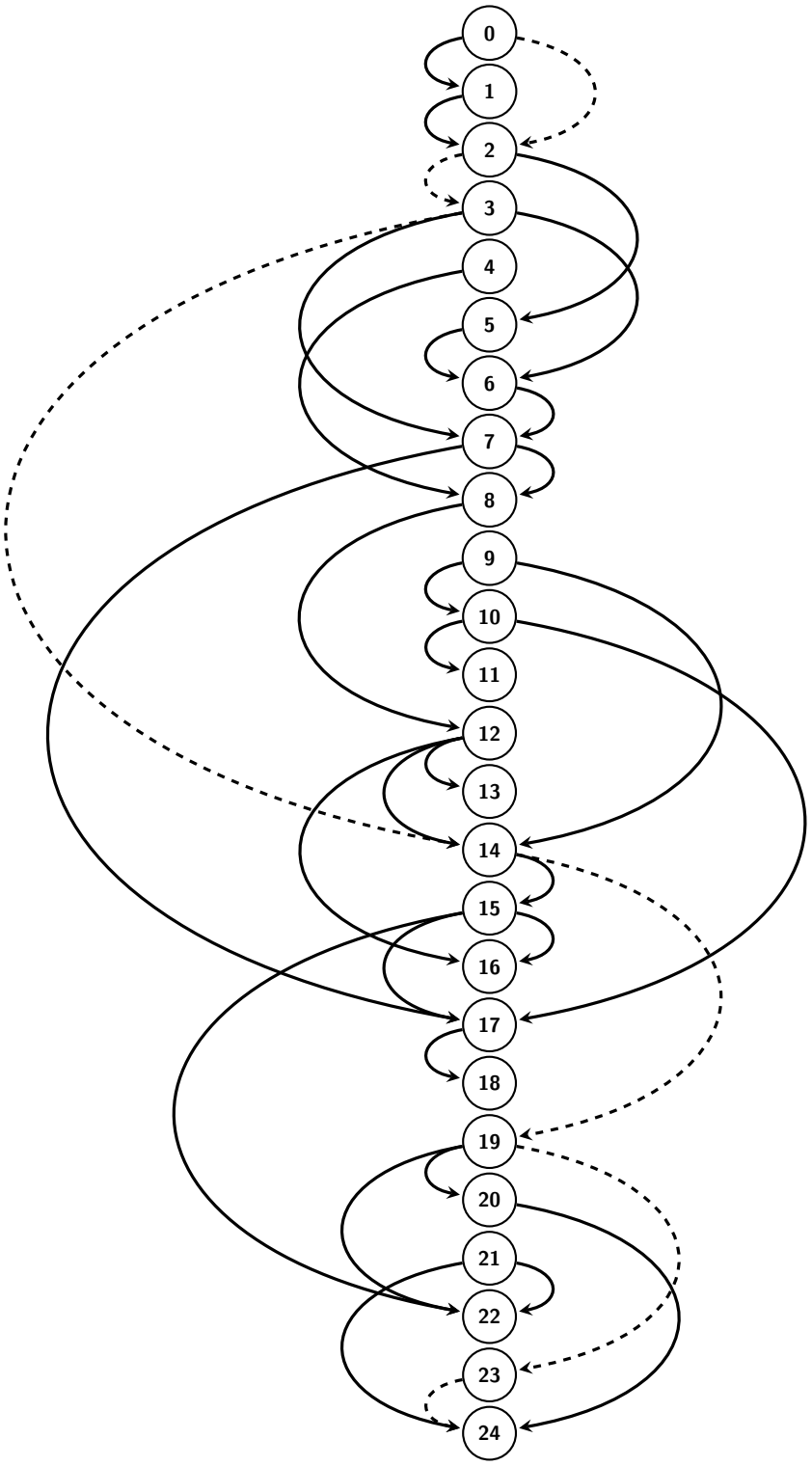

FIGURE 5.9: Graph of maximal palindromes for string from Fig. 5.2 and the shortest path, providing the maximal palindromic factorisation.

5.5 Performance Analysis

We created a basic implementation in order to build some intuition for the performance of the proposed algorithm. The aim of the testing was to determine how the percentage of non-solid symbols in a text influenced the run-time of the algorithm, and the likelihood that a valid palindromic factorisation could be found. The number of non-solid symbols is an important factor to consider when determining efficiency, as such symbols are the primary source of additional algorithmic complexity [89].

Much like the initial tests within Chapter 3, the testing was performed using random data, produced with random numbers generated from `http://random.org` and the associated API [69]. The methodology for creating test data for both tests within Fig. 5.10 and Fig. 5.11 was identical.

For each test iteration, a random text T of length n was generated, with a specific number p of characters representing non-solid symbols and the remaining $n-p$ characters representing solid symbols. Given the desired text length n and the number of non-solid symbols p, the text T of length n was generated by initially forming a string $T = d_0\, d_1\, \ldots\, d_{p-1}\, s_0\, s_1\, \ldots\, s_{n-p-1}$, where d_i and s_i represent randomly chosen non-solid and solid symbols respectively.

Solid symbols s_i were a randomly chosen character from the alphabet $\Sigma = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$ with equal probability. Non-solid symbols were a randomly chosen subset of the power set of Σ with size greater than 1, i.e. a randomly chosen element of $\{\tilde{x} \in \mathcal{P}(\Sigma) : |\tilde{x}| > 1\}$ with equal probability.

The order of the characters in the text T was then randomly shuffled, to produce on average a uniform distribution of non-solid and solid symbols throughout the string. A single test iteration was then performed on this text T, with a new random text T generated with each subsequent iteration of the test for a given set of parameters. The results of the tests are presented in Fig. 5.10 and Fig. 5.11.

Note that due to the particulars of the data structures used within the implementation, the run-time as shown in Fig. 5.10, could potentially be improved by incorporating more efficient data structures. However, the analysis of the success rate of finding a factorisation as shown in Fig. 5.11, is independent of the particular implementation used, and accurately reflects the capabilities of the algorithm.

For each set of test parameters in Fig. 5.10, 10,000 iterations were performed, with each iteration using a new randomly generated text. The average run-time across all iterations was taken as the representative for that particular test configuration.

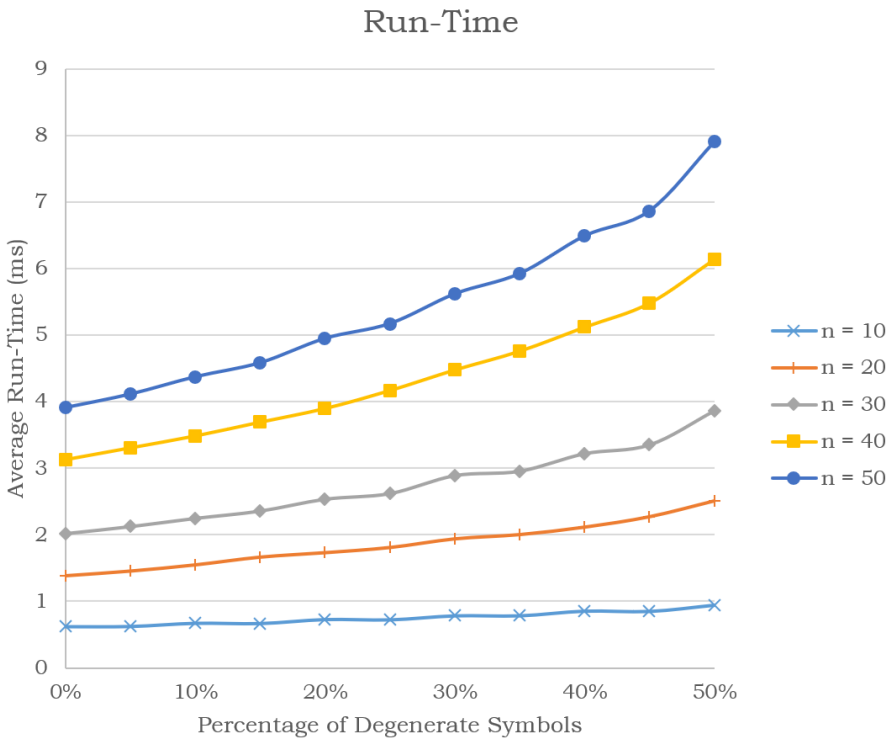

FIGURE 5.10: Average algorithmic run-time to attempt to find a palindromic factorisation in a random text of various lengths and with various ratios of non-solid to solid symbols. n is the length of the text.

As would be expected, the run-time of the algorithm may be increased by either increasing the number of non-solid symbols or by increasing the length n of the text. Based on this sample of testing over the DNA alphabet, a text with 50% non-solid symbols requires approximately twice the run-time to analyse as a purely solid text, regardless of the text length.

The run-time appears to increase super-linearly with an increase in non-solid symbols, though this effect is less visible for shorter texts.

Due to limitations in the initial testing implementation, the length n of the text was limited to 50 characters. It would prove interesting to create an alternative implementation of the proposed algorithm, capable of processing larger text lengths, to determine if the run-time patterns seen in Fig. 5.10 are representative of the algorithm for a wider range of input parameters.

For each set of test parameters in Fig. 5.11, 10,000 iterations were performed, with each iteration using a new randomly generated text. The total number of texts with factorisations found out of the total 10,000 attempted, was then converted to a success percentage.

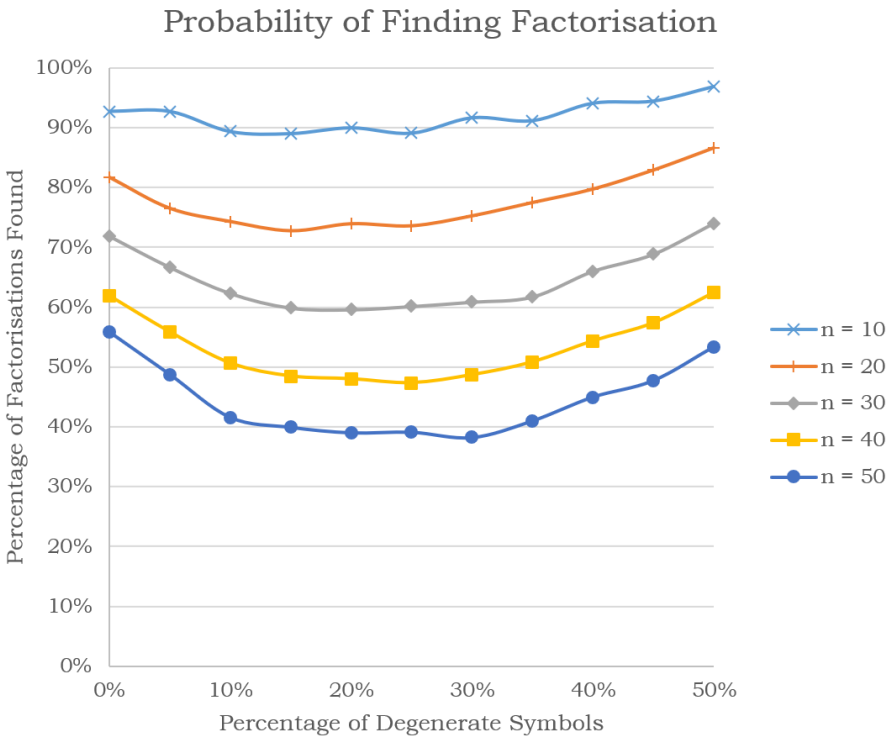

FIGURE 5.11: Estimated probability of finding a palindromic factorisation in a random text of various lengths and with various ratios of non-solid to solid symbols. n is the length of the text.

Although we assumed that increasing the number of non-solid symbols should increase the possibility of a successful factorisation being found, the data shows this is not the case. For each text length n, it appears that a minimum likelihood of successful factorisation occurs with approximately 20-30% non-solid symbols, with an increased probability below and above this figure.

This pattern becomes more pronounced for longer texts, and seems to indicate that a text with 50% non-solid symbols has the same probability of the algorithm finding a factorisation as a purely solid text, regardless of the text length.

It was not determined why this particular pattern of successes was found within initial testing, and it might represent either limitations in the range of testing, deficiencies in the algorithm or deficiencies in the libraries used for the data structures within the implementation. Further exploration of this phenomenon may represent potential for further research.

5.6 Conclusion

An algorithm has been presented for finding all maximal palindromes of a degenerate string, modelled as a degenerate string $\tilde{x}$ of length n with no more than k non-solid symbols. Additionally, we demonstrate how to determine a maximal palindromic factorisation. The proposed algorithm makes use of a suffix tree data structure, in addition to a matching table for the purpose of comparing non-solid symbols. The algorithm requires $\mathcal{O}(k|\Sigma|(k+\log|\Sigma|)+kn)$ time and $\mathcal{O}(k(k+|\Sigma|)+n)$ space. Additionally, under some reasonable assumptions on k and Σ we may further reduce the theoretical complexity bound. For instance, if we know that $k \leq \log(n)$ and treat $|\Sigma|$ as a small constant (e.g. the DNA alphabet), then the required time and space becomes $\mathcal{O}(n\log(n))$ and $\mathcal{O}(n)$ respectively.

A possible improvement would be to remove the restriction that all palindromes in the factorisation are maximal. If ordinary palindromes were permitted, this would necessarily change the approach currently being used, and likely require a higher algorithmic complexity to determine a valid factorisation. Additionally, permitting small gaps between consecutive factors would also improve the practicality of the algorithm with regards to the analysis of genetic data. Either of these improvements would prove significant and non-trivial.

We implemented these algorithms and subsequently tested their efficacy, however they were later generalised to determine inverted repeats in degenerate strings, an even more general case of maximal palindromes. The details of this work are further explored in Chapter 6, where we apply the searching of maximal palindromes to the problem of identifying inverted repeats within real IUPAC-encoded DNA data. This chapter serves as a precursory study of palindromes in degenerate strings, in preparation for more general algorithms, their implementations and associated results, as described in the following chapter on the topic of inverted repeats.

Chapter 6

Inverted Repeats

The International Union of Pure and Applied Chemistry (IUPAC) encoding, provides a systematic way to describe DNA sequences that allow for multiple possible nucleotides to occur at a single location within the sequence [90]. An inverted repeat is a sequence of nucleotides followed downstream by its reverse complement, potentially with a gap in the centre [91]. In this chapter, we present a software implementation that is capable of identifying inverted repeats with gaps and possible mismatches, according to the IUPAC matching scheme, and show that our application compares favourably to a similar application packaged with EMBOSS [20, 21]. We demonstrate that our software IUPACpal identifies previously unidentified inverted repeats when compared with EMBOSS, and may perform this task with orders of magnitude improved speed.

6.1 Background

Finding regularities in DNA sequences is an essential component of numerous applications of molecular biology [92, 93]. As previously stated in Chapter 5, computational biology relies heavily on developing algorithmic methods to study regularities such as repeats, covers and indeed palindromes [94, 95]. However DNA data is often not expressed in certain terms, using the primary 4 characters. Data is often expressed as IUPAC-encoded sequences, as specified by the The International Union of Pure and Applied Chemistry (IUPAC) [96–98].

As defined in Section 5.2.1, the term *degenerate* string or *indeterminate* string refers to a generalised string in which individual characters may have multiple values. IUPAC strings may be considered as a more precise variant of degenerate strings, limited to characters representing subsets of the primary DNA alphabet.

One particular set of genetic structures of interest are *inverted repeats*. These structures are closely connected to palindromes, and are in fact equivalent when no gap within the structure is present [99]. Inverted repeats describe a formation that serves various biological functions. For example, in delineating boundaries of transposons and highlighting regions capable of self-complementary base pairing [100]. Such properties often play a role in the study of mutation and disease [101].

Software has already been developed, for the purposes of identifying inverted repeats within IUPAC-encoded sequences [102–104]. Throughout this chapter we make reference to one such example, namely the widely used PALINDROME application within The European Molecular Biology Open Software Suite (EMBOSS) [20, 21]. The contribution of this work is to present a method of identifying additional potential inverted repeats which are currently ignored by EMBOSS, in addition to providing a significant real-world speed improvement.

6.2 Problem Outline

6.2.1 Terminology

As stated previously, for a given string x, we use the notation x^R to refer to the reversal of x. A *palindrome* is a string p which is equal to its reversal i.e. $p = p^R$.

We further use the notation $\bar{x}$ to refer to the *complement* of a string x, where the complement is defined by some bijective function $f : \Sigma \to \Sigma$. In the case of a DNA alphabet, the natural choice of a complement function over the alphabet $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ is the function with the bijective mappings $\mathtt{A} \longleftrightarrow \mathtt{T}$ and $\mathtt{C} \longleftrightarrow \mathtt{G}$ [105]. For a given bijective function f, the complement string $\bar{x}$ of a string x is such that $\bar{x}[i] = f(X[i])$ for all i.

Closely related to palindromes, we define an *inverted repeat* as a string that can be expressed in the form $w\bar{w}^R$ for some string w. We may generalise inverted repeats by allowing a central gap, which we call a *gapped inverted repeat*. A gapped inverted repeat is therefore a string that can be expressed in the form $wg\bar{w}^R$ for some pair of strings w and g.

Finally, we may introduce mismatches by permitting the two occurrences of w within $wg\bar{w}^R$ to differ by some number of characters, i.e. some Hamming distance. We refer to a string as a *gapped inverted repeat within k mismatches* when it can be expressed in the form $wg\bar{w}^R$ with $\delta_H(w, \bar{w}^R) \leq k$. In the remainder of this chapter, we use the term

inverted repeat irrespective of whether it contains a gap, unless making the distinction is necessary.

An example of an inverted repeat, which makes use of a gap and mismatches is shown in Fig. 6.1. This illustrates the most commonly used diagrammatic representation of inverted repeats.

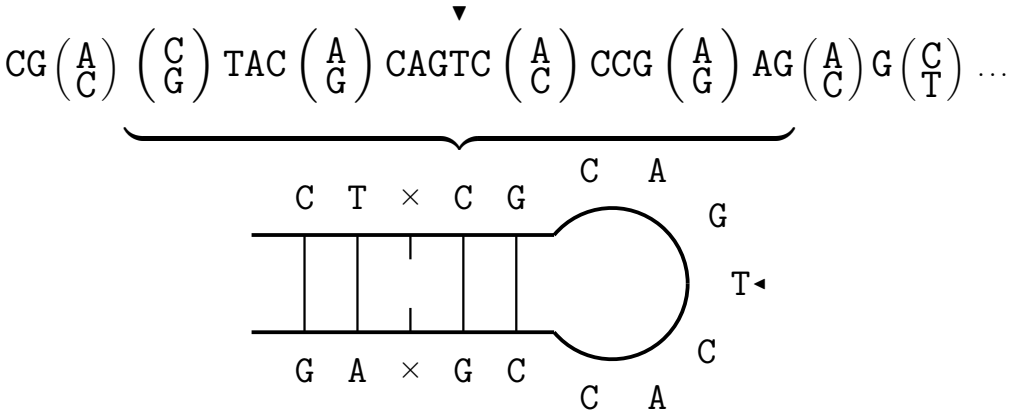

FIGURE 6.1: Example of an inverted repeat with a gap of 7 and 1 mismatch.

The degenerate string representation of an inverted repeat shown in Fig. 6.1 may be represented as a IUPAC-encoded string, which is demonstrated in Fig. 6.2.

CGMSTACRCAGTCMCCGRAGMGY ...

FIGURE 6.2: Example of a IUPAC-encoded string.

Using the IUPAC encoding we may consider uncertain DNA strings as solid strings, rather than degenerate, despite the fact that a IUPAC-encoded string fundamentally represents a degenerate string over the alphabet {A,C,G,T}.

6.2.2 Problem Statement

We now formally define the problem addressed within this chapter, namely Identification of Inverted Repeats with Gaps and Mismatches:

IDENTIFICATION OF INVERTED REPEATS WITH GAPS AND MISMATCHES

Input:
IUPAC-encoded string x of length n, a natural number k representing the maximum number of permitted mismatches, a pair of natural numbers m and M representing the minimum and maximum length respectively of the identified inverted repeats, and a natural number g specifying the maximum permitted gap size.

Output:
Array of inverted repeats each represented by 4 indexes (a, b, c, d) such that for some string $w \equiv_k x[c\,..\,d]$ it is that case that $x[a\,..\,b]$ is a complement of w. Additionally, the gap $c - b - 1 \leq g$ and the length of the inverted repeat is within the stated bounds i.e. $m \leq b - a + 1 \leq M$ and $m \leq d - c + 1 \leq M$.

6.3 Solution

6.3.1 Tools

The International Union of Pure and Applied Chemistry (IUPAC) encoding is an extended alphabet Σ^+ of symbols [90], which provides a single symbol representation for every one of the 15 possible non empty subsets of the standard 4 character alphabet $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. For example, the symbol `B` represents the set {`C`,`G`,`T`}. This encoding provides a natural way to represent degenerate symbols using single characters. The standard set of IUPAC symbols is $\Sigma^+ = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}, \mathtt{R}, \mathtt{Y}, \mathtt{S}, \mathtt{W}, \mathtt{K}, \mathtt{M}, \mathtt{B}, \mathtt{D}, \mathtt{H}, \mathtt{V}, \mathtt{N}\}$. The symbol `U` is also sometimes used instead of `T`, and the symbol `*` instead of `N` [106]. We therefore treat these two pairs interchangeably.

This raises the question of how to determine complements of such IUPAC symbols, extending the current matching scheme $\mathtt{A} \longleftrightarrow \mathtt{T}$, $\mathtt{C} \longleftrightarrow \mathtt{G}$ over Σ to the full IUPAC alphabet Σ^+.

The current PALINDROME application within the EMBOSS package uses a method by which every IUPAC symbol is assigned a single unique complement, by first taking complements of the underlying characters of the represented subset of Σ. For example, the complement of $\mathtt{B} = \{\mathtt{C}, \mathtt{G}, \mathtt{T}\}$ is $\mathtt{V} = \{\mathtt{G}, \mathtt{C}, \mathtt{A}\}$, and therefore $\mathtt{B} \longleftrightarrow \mathtt{V}$. We dub this method *simple complement matching*.

However, if we choose to interpret IUPAC symbols as representing a set of possibilities for which there is a single real value, then this type of matching does not take into account all possible match scenarios. Consider for example the symbol $\mathtt{R} = \{\mathtt{A}, \mathtt{G}\}$ when

compared with the symbol $\mathtt{C} = \{\mathtt{C}\}$. Under simple complement matching, the symbols `R` and `C` do not match, despite the fact that `R` contains `G`, the complement of `C`.

Because of this potential shortcoming, we define *degenerate complement matching* over the IUPAC alphabet. Under this matching scheme, two IUPAC symbols $\mathtt{I}_1$ and $\mathtt{I}_2$ match if and only if there exists a pair of characters $\sigma_1 \in \mathtt{I}_1$ and $\sigma_2 \in \mathtt{I}_2$ such that $\sigma_1 \longleftrightarrow \sigma_2$.

We present a visualisation of both the simple and degenerate complement matching schemes in Fig. 6.3 and Fig. 6.4 respectively, where white blocks indicate mismatch and filled blocks indicate match. Note that when considering sequences exclusively over the alphabet $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, there is no distinction between simple and degenerate complement matching.

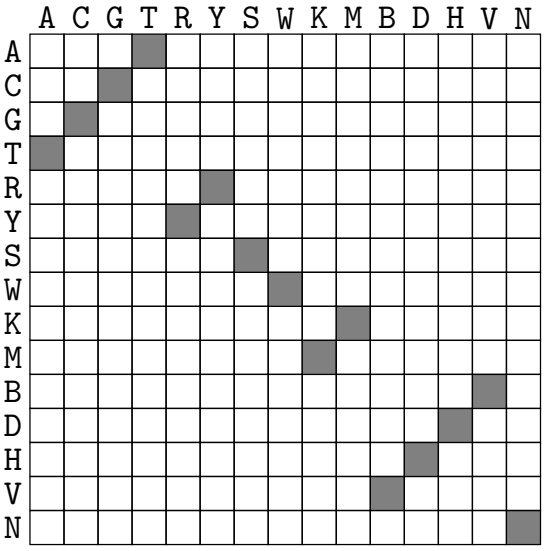

FIGURE 6.3: Simple complement matching. Filled box indicates a match.

Simple complement matching limits the possible number of matches significantly. IUPAC characters which represent a large number of possibilities do not have an increased likelihood of matching other characters. Under this matching scheme, every IUPAC character matches uniquely with precisely one other IUPAC character, and so may be treated as a bijective function.

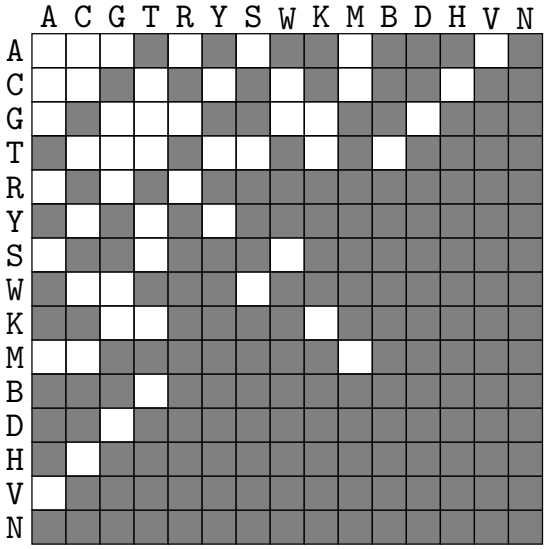

FIGURE 6.4: Degenerate complement matching. Filled box indicates a match.

Degenerate complement matching considers a match to occur whenever two IUPAC characters have any possibility of matching between any of their respective associated subsets of characters. The variety of possible matches is much larger, though note that the primary characters {A,C,G,T} still match identically in comparison to simple complement matching. IUPAC characters which represent a large number of possibilities have an increased likelihood of matching other characters. Under this matching scheme, there is not a unique complement for each IUPAC character, and therefore an associated bijective function may not be defined. Therefore, a string w may have a set of associated complements $\bar{w}$, and an inverted repeat $wg\bar{w}^R$ may take several acceptable forms for the same string w.

6.3.2 Algorithm

For the problem addressed in this particular chapter, the focus was on the software implementation above the underlying algorithm. The algorithm used was extended from the maximal palindrome identifying algorithm from Chapter 5. Effort was made to ensure space and time efficiency within the specific components of the implementation, and additional steps were included to address gaps and mismatches through the use of the kangaroo method of common extensions [107, 108]

Within these additional steps, the algorithm exhaustively identifies all inverted repeats by examining each position within a sequence and determining every valid inverted repeat with its centre at that position which adheres to the given input parameters.

For a given centre, the possible inverted repeats are determined by first identifying symbols which are equidistant from the centre and are considered to mismatch. As was done in Chapter 5, this is performed by the kangaroo method, which therefore requires preprocessing the sequence to obtain both its suffix tree and the suffix tree of the reverse of the sequence.

Given that these mismatches can be identified, the procedure for finding inverted repeats considers a minimal initial gap which is subsequently increased in order to reduce the number of mismatches inside the inverted repeat being considered, and thus permits a longer extension (inspect Fig. 6.5).

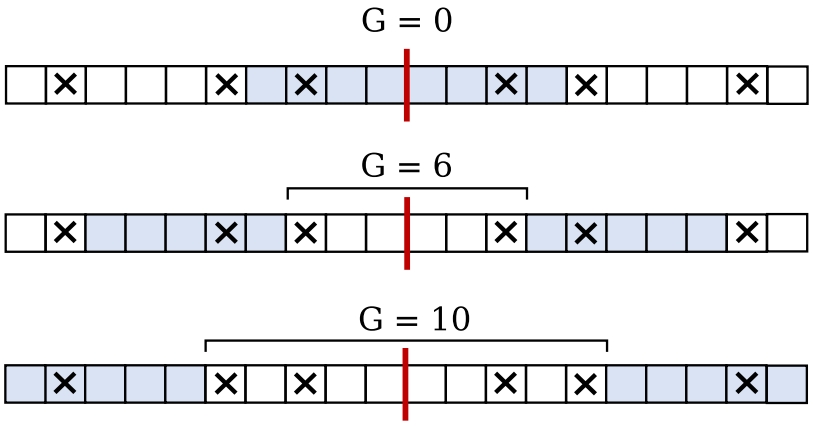

FIGURE 6.5: Inverted repeats in a sequence for a given centre with 1 permitted mismatch. The centre is marked in red. The size of the gap is given by G. Mismatching symbols are marked with the symbol $\times$. The inverted repeat is indicated by the shaded cells.

This demonstrates the principle of finding several unique inverted repeats with the same centre by extending the gap to effectively swallow an additional mismatch, such that the inverted repeat may be extended to the position directly adjacent to the next mismatch. This extending procedure is performed repeatedly to obtain all inverted repeats for a given centre, while taking into account the parameters specifying the maximum gap and the size range for the inverted repeat itself. The algorithm maintains efficiency by calculating only the necessary mismatch locations needed for a given set of parameters, and no more.

For full details on the implementation and underlying algorithm, the original source code is made available at the following link:

`https://github.com/steven31415/IUPACpal`

6.4 Implementation

6.4.1 Interface

The software IUPACpal was implemented to mimic the workflow, parameters and output format of the competitor EMBOSS, to better enable direct comparisons in performance [20]. By making the key features similar and output format identical, this also minimises the learning curve of using the software for those familiar with using EMBOSS. The application requests the following parameters:

INPUT FILE (0)
SEQUENCE NAME (1)
OUTPUT FILE (2)
MINIMUM LENGTH (3)
MAXIMUM LENGTH (4)
MAXIMUM GAP (5)
MAXIMUM MISMATCHES (6)

IUPACpal is run via a command-line interface with the following command:

```
$ ./IUPACpal -f <0> -s <1> -o <2> -m <3> -M <4> -g <5> -x <6>
```

Note that the parameters `m`, `M`, `g` and `x` are optional. By default there are no limits on the inverted repeat size, and no gap or mismatches permitted. The options are provided to improve running time, when specific features of the desired inverted repeats are known.

The resulting output file is given in an identical format to that of EMBOSS, in which all the discovered inverted repeats are identified by their index locations (1-based indexing) alongside their character representation. An example of this output representation as applied to the inverted repeat from Fig. 6.1 is shown below:

```
 4  STACR  8
    || ||
20  GARGC  16
```

The inverted repeats found by both tools are maximal, i.e. can be extended no further (unless further mismatches are utilised). Additionally, the leftmost and rightmost symbol in any reported inverted repeat must necessarily match.

6.4.2 Performance Analysis

Several performance tests were run, providing the PALINDROME tool from EMBOSS and IUPACPAL the same input data, and considered both their respective run-times and numbers of inverted repeats found. We performed all tests on contiguous subsequences of the human X chromosome with IUPAC characters, taken from reference genome GRChg37. For consistency with EMBOSS, inverted repeats are interchangeably referred to as palindromes within the figures below.

The first test was used to determine and compare the number of inverted repeats found. A script was written to determine whether or not IUPACPAL and EMBOSS found the same palindromes, and it was determined that IUPACPAL could find all the same palindromes as EMBOSS in addition to further palindromes or more maximal extensions of palindromes found by EMBOSS. On further analysis this appeared to be not only a result of the alternate complement matching system, but also a result of the difference in implementation between the two pieces of software.

A comparison of the number of inverted repeats for different mismatches is shown in Fig. 6.6. It was determined that changing the number of mismatches had the greatest influence on the number of inverted repeats found, and therefore this was the parameter that was varied.

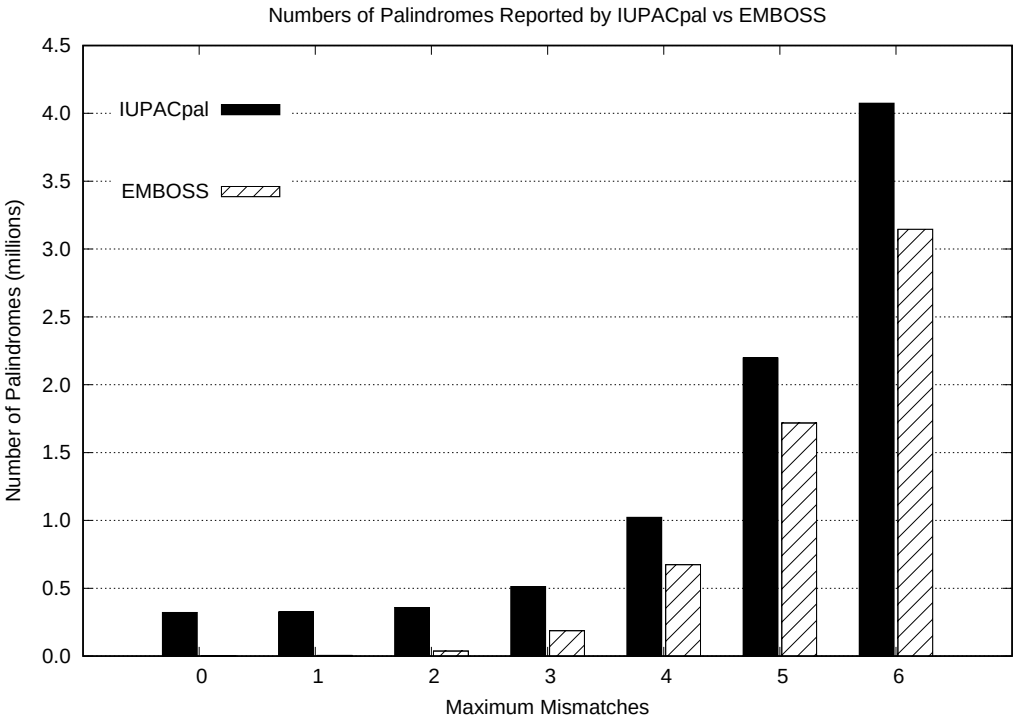

FIGURE 6.6: Comparison of palindromes found over 1,000,000 DNA characters. MINIMUM LENGTH: 10. MAXIMUM LENGTH: 100. MAXIMUM GAP: 100.

The results displayed in Fig. 6.6 show that IUPACpal consistently found more inverted repeats than EMBOSS. For a low number of mismatches, there is an extreme difference, though the ratio between the two reduces as the number of permitted mismatches increases.

The next consideration was the run-time of both systems. Initially only the number of mismatches was changed, and the affect this had on run-time is shown in Fig. 6.7.

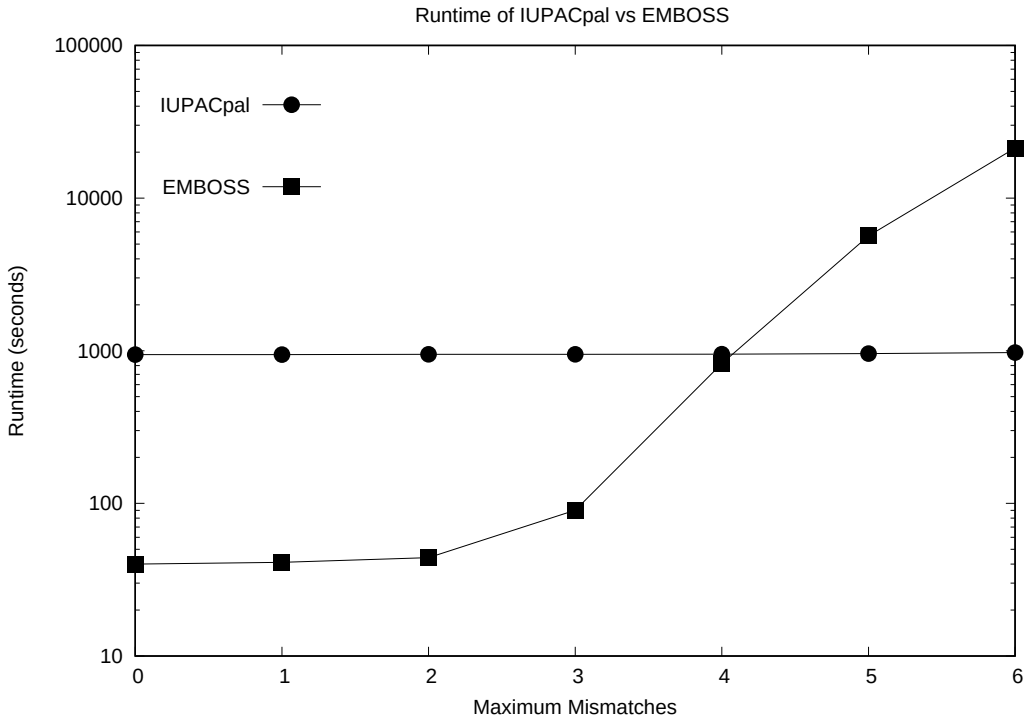

FIGURE 6.7: Comparison of run-time over 1,000,000 DNA characters. MINIMUM LENGTH: 10. MAXIMUM LENGTH: 100. MAXIMUM GAP: 100.

Note that the run-time is plotted on a logarithmic scale, due to the extreme differences in speed. It appeared that for a small number of mismatches, specifically 3 or less, the competitor EMBOSS performed faster. However with a greater number of mismatches, the speed of EMBOSS rapidly increases in a non-linear fashion, whereas IUPACpal retains a speed on the order of 1000 seconds. Naturally IUPACpal also requires more time to execute for a greater number of mismatches, but the vast difference in orders of magnitude when compared to the competitor, make this gradual increase relatively imperceptible.

Finally, a further run-time test was performed, in which both the number of mismatches and the size of the gap within the inverted repeat were varied. The results are plotted on a pair of heat-maps in Fig. 6.8 and 6.9, in which darker colours represent a faster execution time, using a logarithmic colour scale.

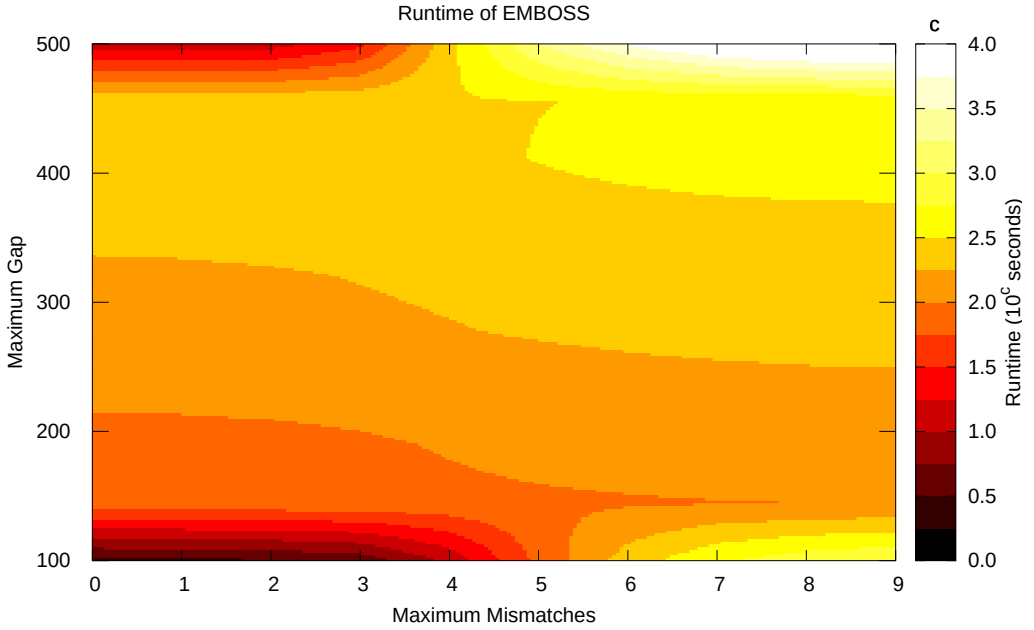

FIGURE 6.8: Comparison of run-time over 1,000,000 DNA characters. MINIMUM LENGTH: 10. MAXIMUM LENGTH: 100. MAXIMUM GAP: 100.

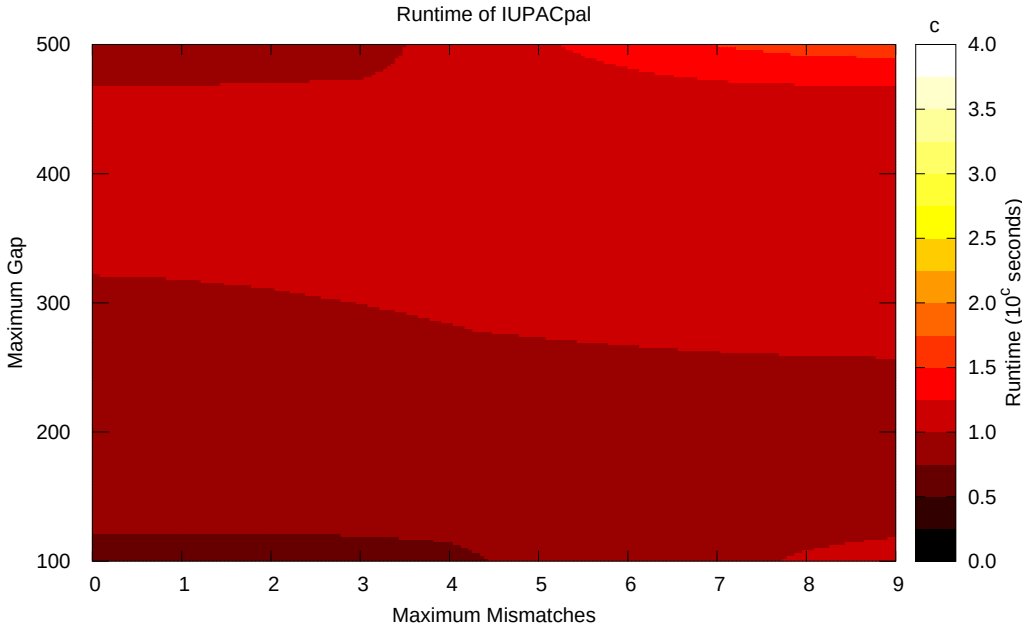

FIGURE 6.9: Comparison of run-time over 1,000,000 DNA characters. MINIMUM LENGTH: 10. MAXIMUM LENGTH: 100. MAXIMUM GAP: 100.

6.5 Conclusion

During the research process, the initial aim was to implement a variation of the popular EMBOSS tool, that could find inverted repeats using an alternative complement matching system, in order to identify a larger number of potential inverted repeats within IUPAC-encoded sequences.

However in carrying out this research, there was a surprising realisation that the implementation of EMBOSS was far from efficient, with the IUPACpal tool being orders of magnitude faster. Though the initial goal of the tool is still achieved, the most exciting revelation is the far more favourable execution time of the IUPACpal tool.

Because the data pipeline and workflow were intentionally created to mimic that of EMBOSS, it would appear that this new tool may provide a valuable new resource for computational biologists. The source code is publicly available for use, as noted in Section 6.3.2, and may be freely modified accordingly.

Future work could involve further developing the tool to incorporate additional functionality which is already native to the EMBOSS tool, such as the ability to handle multiple DNA data file formats [109, 110]. With some additional development towards usability, the IUPACpal tool would represent a significant improvement over the EMBOSS tool in both theory and practice.

Chapter 7

Closed Strings

A closed string contains a proper factor occurring as both a prefix and a suffix but not elsewhere in the string. Closed strings were initially introduced as objects of combinatorial interest. This chapter addresses a new problem by extending the closed string problem to the k-closed string problem, for which a level of approximation is permitted up to a number of Hamming distance errors, set by the parameter k.

This chapter addresses the problem of deciding whether or not a given string of length n over an integer alphabet is k-closed and additionally specifying the border resulting in the string being k-closed. Specifically, an $\mathcal{O}(kn)$-time and $\mathcal{O}(n)$-space algorithm to achieve this along with the pseudocode of an implementation and proof-of-concept experimental results.

7.1 Background

A bordered string x is such that there exists a prefix of x which is also a suffix of x. A closed string (or a closed word) is a bordered string that satisfies an additional property: the border does not occur elsewhere in the string. There are a number of earlier studies dealing with closed strings. Fici et al [111] introduced the notion of closed strings in addition to characterisations of this class.

The more practical relevance of closed strings was established via their relationship with palindromic strings. The number of closed factors in a string is minimised if these factors are also palindromic [112]. Additionally it has been shown that the upper bound on the number of palindromic factors of a string coincides with the lower bound on the number of closed factors [113]. Thus the study of closed strings shows potential applications in connection with applications of palindromes [86]. On the algorithmic side, Badkobeh

et al presented (among others) an algorithm for the factorisation of a given string of length n into a sequence of longest closed factors in time and space $\mathcal{O}(n)$ and another algorithm for computing the longest closed factor starting at every position in the string in $\mathcal{O}(n\frac{\log n}{\log\log n})$ time and $\mathcal{O}(n)$ space [112].

Here we extend the definition of closed strings to k-closed strings, for which a level of approximation is permitted up to a number of Hamming distance errors, set by the parameter k. The main contribution of this chapter is an $\mathcal{O}(kn)$-time and $\mathcal{O}(n)$-space algorithm for deciding whether or not a given string of length n over an integer alphabet is k-closed. The border that results in the string being k-closed is also specified.

This chapter makes use of the kangaroo method for longest common extensions with mismatches [107, 108] and presents the possible methods of generalising the closed string problem to the k-closed string problem. This includes a detailed implementation of the algorithmic solution in addition to proof-of-concept experimental results.

7.2 Problem Outline

7.2.1 Terminology

If a string b is both a proper prefix and a proper suffix of a non-empty string x, then b is called a *border* of x. A string x is said to be *closed* if and only if it is empty or if there exists a border b of x that occurs exactly twice in x (i.e. only as a prefix and suffix). In other words, b satisfies the following 2 conditions:

(1) $b = x[0\,..\,|b|-1] = x[|x|-|b|\,..\,|x|-1]$

(2) $b \neq x[i\,..\,i+|b|-1]$, for all $1 \leq i \leq |x|-|b|-1$

If x is closed, we call such a b the *closed border* of x. We additionally define the special case of a single letter $a \in \Sigma$ to be closed, with the empty string ε as the border of a.

For instance, the string `ACA` is closed, since the factor `A` occurs only as a prefix and as a suffix. The string `ACAA`, on the contrary, is not closed: `A` has an internal occurrence.

The definition of closed strings can be generalised to *k-closed strings*, where k expresses a Hamming distance error bound. This is useful for dealing with strings where errors or approximations in the data may occur.

In order to define k-closed strings, it proves useful to initially introduce a simpler definition which we dub *k-pseudo-closed strings* in Def 7.1.

Definition 7.1. A string x of length n is called *k-pseudo-closed* if and only if $n \leq 1$ or the following properties are satisfied:

1. There exists some proper prefix u of x and some proper suffix v of x of length $|u| = |v|$, such that $\delta_H(u, v) \leq k$.

2. Except for u and v, there exists no factor w of x of length $|w| = |u| = |v|$ such that $\delta_H(u, w) \leq k$ or $\delta_H(v, w) \leq k$.

We call such a pair u and v the *k-pseudo-closed border* of x. In the case where $n \leq 1$, we assign ε as the k-pseudo-closed border.

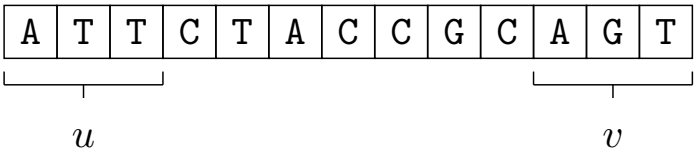

FIGURE 7.2: Example of k-pseudo-closed string for $k = 1$.

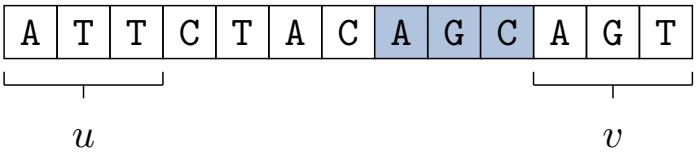

FIGURE 7.3: Example of non-k-pseudo-closed string for $k = 1$.

With k-pseudo-closed strings defined, we may now define the more general case of k-closed strings.

Definition 7.4. A string x of length n is called *k-closed* if and only if $n \leq 1$ or the following properties are satisfied for some k' where $0 \leq k' \leq k$:

1. There exists some proper prefix u of x and some proper suffix v of x of length $|u| = |v|$, such that $\delta_H(u, v) \leq k'$.

2. Except for u and v, there exists no factor w of x of length $|w| = |u| = |v|$ such that $\delta_H(u, w) \leq k'$ or $\delta_H(v, w) \leq k'$.

For the above definition, the pair u and v for the smallest k' is called the *k-closed border* of x. In the case where $n \leq 1$ we assign ε as the k-closed border.

This definition of k-closed strings permits us to trivially conclude the following crucial lemmas:

Lemma 7.5. *x is k-closed $\iff$ $\exists k'$ where $0 \leq k' \leq k$, such that x is k'-pseudo-closed.*

Lemma 7.6. *A string x that is k-closed is also r-closed for all $r > k$.*

This demonstrates that a smaller value of k corresponds to the designation of k-closed being a stronger statement on the properties of x.

It additionally follows from Def. 7.4 that the k-closed border of a string x is unique by Lem. 7.7.

Lemma 7.7. *A length-n k-closed string x, $n > 1$, has exactly one k-closed border, i.e there exists exactly one prefix u and one suffix v satisfying the conditions in Def. 7.4 for the smallest $k' \leq k$.*

Proof. Since x is k-closed, it has at least one k-closed border and an associated smallest $k' \leq k$ for which the conditions are satisfied. Let us consider the longest of these k-closed borders, and call u and v the prefix and suffix respectively, comprising the longest k-closed border with length $|u| = |v|$. Let us assume a second k-closed border exists, comprised of the prefix and suffix, u' and v' respectively. We know that $|u'| = |v'| < |u| = |v|$ and $u' = u[0\,..\,|u'| - 1]$. Since $u \equiv_{k'} v$ it is trivially true that $u[0\,..\,|u'| - 1] \equiv_{k'} v[0\,..\,|u'| - 1]$ and therefore $u' \equiv_{k'} v[0\,..\,|u'| - 1]$. Thus we see that u' k'-matches the prefix of v of the same length, and this corresponds to an occurrence of u' within x, i.e. $u' \equiv_{k'} x[n - |v|\,..\,n - |v| + |u'| - 1]$, where n is the length of x, which is an internal occurrence of u' in x. We arrive at a contradiction due to Condition 2 of Def. 7.4 being violated, therefore no second k-closed border can exist. □

Let x be a non-empty k-closed string of length n. The following properties follow easily from Def. 7.4 and Lem. 7.7:

1. x has exactly one k-closed border.

2. If $n > 1$, there exists a string w with $|w| < n$ and a natural number k', with $0 \leq k' \leq k$, such that $w \equiv_{k'} x[i\,..\,i + |w| - 1]$ for exactly two values of i and no others; specifically $i = 0$ and $i = n - |w|$.

3. There exists a natural number k', with $0 \leq k' \leq k$, such that the longest repeated prefix (resp. suffix) of x within k' errors, is equal to u (resp. v), where u and v are the prefix and suffix, respectively, comprising the k-closed border.

4. There exists a natural number k', with $0 \leq k' \leq k$, such that any repeated prefix (resp. suffix) of x within k' errors is necessarily a prefix (resp. suffix) of u (resp. v), where u and v are the prefix and suffix, respectively, comprising the k-closed border.

Note that k-closed strings are a generalisation of the standard closed string definition, which may now alternatively be referred to as 0-closed strings in this context.

We display both an example and counterexample of a 1-closed string in Fig. 7.8 and Fig. 7.9. Note that for the string `GTGAGTGGTA` we illustrate only that a border length of 3 with error 1 is not a possible 1-closed border. To fully verify it is non 1-closed, all combinations of border lengths and error levels $0 \leq k' \leq 1$ must be considered. It is in fact possible to show that no borders of any length exist that satisfy the 1-closed criteria, therefore the string is indeed non 1-closed (and by Lem. 7.6 also non 0-closed).

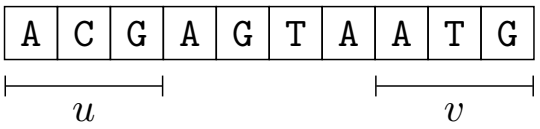

FIGURE 7.8: Example of k-closed string for $k = 1$.

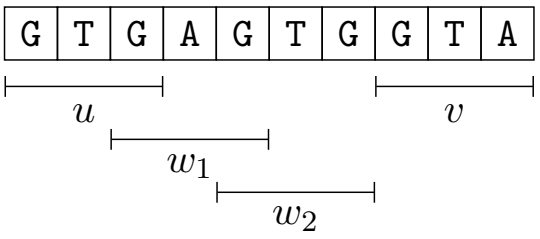

FIGURE 7.9: Example of non-k-closed string for $k = 1$.

7.2.2 Problem Statement

We now formally define the problem addressed within this chapter, namely Identification of k-Closed Borders:

IDENTIFICATION OF k-CLOSED BORDERS

Input:
A string x of length n and a natural number k, $0 < k < n$.

Output:
The k-closed border or -1 if x is not k-closed.

As a novel definition, we begin with the basic task of identifying the borders of this new structure as a starting point.

7.3 Solution

7.3.1 Tools

In addition to the definition of k-closed strings, we further document some alternative definitions of Def. 7.4 which proved useful in obtaining the main result.

Definition 7.10. A string x of length n is called *k-weakly-closed* if and only if $n \leq 1$ or the following properties are satisfied:

1. There exists some proper prefix u of x and some proper suffix v of x of length $|u| = |v|$, such that $\delta_H(u, v) \leq k$.

2. Both factors u and v occur only as a prefix and suffix respectively within x, i.e. no internal occurrences of u or v exist in x.

We call such a pair u and v a *k-weakly-closed border* of x. In the case where $n \leq 1$, we assign ε as the k-weakly-closed border.

Def. 7.10 is satisfied in situations where the border may have errors, but internal occurrences are considered to not have errors.

For example, `ACTGTAATTAGT` is 1-weakly-closed with a 1-weakly-closed border (`ACT`, `AGT`) of length 3, whereas `ACTGTAGTTAGT` is not 1-weakly-closed. This is shown in Fig. 7.11 and Fig. 7.12.

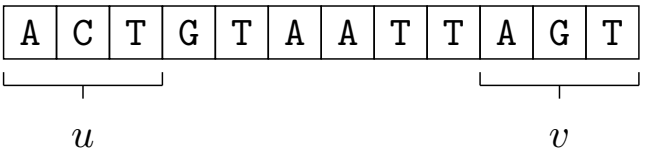

FIGURE 7.11: Example of k-weakly-closed string for $k = 1$.

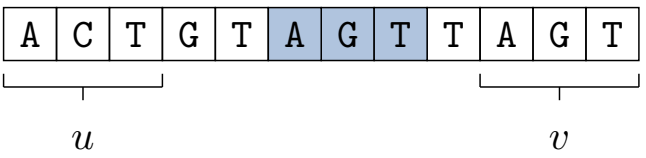

FIGURE 7.12: Example of non-k-weakly-closed string for $k = 1$.

Note that under this definition there may be multiple k-weakly-closed borders.

For example, `TATAGAACATAT` is 2-weakly-closed with two 2-weakly-closed borders (`TAT`, `TAT`) of length 3 and (`TATAG`, `CATAT`) of length 5.

Definition 7.13. A string x of length n is called *k-strongly-closed* if and only if $n \leq 1$ or the following properties are satisfied:

1. There exists some non-empty border b of x.

2. There exists no factor w of x of length $|w| = |b|$ such that $\delta_H(b, w) \leq k$, except the prefix and suffix of x.

We call b the *k-strongly-closed border* of x. In the case where $n \leq 1$, we assign ε as the k-strongly-closed border.

Def. 7.13 is satisfied in situations where the border does not have errors, but internal occurrences may have errors. For example, `ACTGTATCAACT` is 1-strongly-closed with a 1-strongly-closed border `ACT` of length 3, whereas `ACTGTATTAACT` is not 1-strongly-closed. This is shown in Fig. 7.14 and Fig. 7.15.

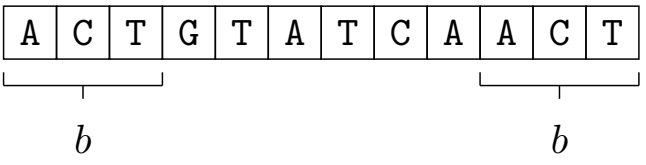

FIGURE 7.14: Example of k-strongly-closed string for $k = 1$.

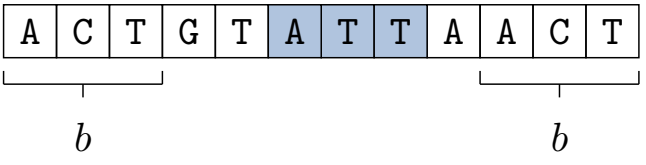

FIGURE 7.15: Example of non-k-strongly-closed string for $k = 1$.

Note that under this definition there is only one k-strongly-closed border.

Condition 1 and Condition 2 of Def. 7.1 may be regarded as a merge of Def. 7.10 and Def. 7.13. Both the border and internal occurrences may have errors. In fact this merging of definitions justifies how Def. 7.1 was chosen as the generalisation of closed strings.

For example, `ATTCTACCGCAGT` is 1-pseudo-closed with a 1-pseudo-closed border (`ATT`, `AGT`) of length 3, whereas `ATTCTACAGCAGT` is not 1-pseudo-closed. This is shown in Fig. 7.2 and Fig. 7.3.

Note that within Def. 7.1, Condition 1 is *less selective* and Condition 2 is *more selective*. Therefore the requirement to satisfy both conditions implies that a 0-closed string is not necessarily k-pseudo-closed and a k-pseudo-closed string is not necessarily 0-closed (hence the *pseudo* term). For instance, `ABAC` is 1-pseudo-closed with a border (`AB`, `AC`), but not 0-closed. In a contrary example, `ABBA` is 0-closed with a border (`A`, `A`) but not 1-pseudo-closed.

7.3.2 Algorithm

We initially begin by constructing the suffix tree of x. As has been discussed in previous chapters, this may be constructed in $\mathcal{O}(n)$ time and space. Recall that once the suffix tree of x is constructed it can be pre-processed within the same complexity to answer any $\text{lce}_k(x, i, j)$ query by applying the kangaroo method in $\mathcal{O}(k)$ time [107, 108].

For the purpose of this algorithm, we draw attention to a specific subset of the possible lce_k queries and store their values in two related data structures. These structures are the *longest prefix k-match array* and *longest suffix k-match array* of string x, denoted by $\text{LPM}_k(x)$ and $\text{LSM}_k(x)$ respectively.

$\text{LPM}_k(x)[j]$ (respectively $\text{LSM}_k(x)[j]$) is defined as the length of the longest factor of x starting (ending) at index j, which matches the prefix (suffix) of x of the same length within k errors, with the exception of the index j corresponding to the prefix (suffix) itself, for which we set a value of -1. Note that within the literature, the LPM array is similar to the *k-prefix table* [114] with the exception of using the -1 flag.

$$\text{LPM}_k(x)[j] = \begin{cases} \max\{l : \delta_H(x[0\,..\,l-1], x[j\,..\,j+l-1]) \leq k\} & j \in [1, n-1] \\ -1 & j = 0 \end{cases}$$

$$\text{LSM}_k(x)[j] = \begin{cases} \max\{l : \delta_H(x[n-l\,..\,n-1], x[j-l+1\,..\,j]) \leq k\} & j \in [0, n-2] \\ -1 & j = n-1 \end{cases}$$

Note that it follows from the definition that the LSM array for a string x is equal to the reverse of the LPM array for the reverse of x, and this logic applies analogously for the prefix array:

$$\mathrm{LSM}_k(x)[j] = \mathrm{LPM}_k(x^R)[n-1-j]$$
$$\mathrm{LPM}_k(x)[j] = \mathrm{LSM}_k(x^R)[n-1-j].$$

Using these identities, we may express the LPM and LSM in terms of the familiar LCE queries, making it possible to apply the kangaroo method to construct them:

$$\begin{aligned} \mathrm{LPM}_k(x)[j] &= \mathrm{lce}_k(x, 0, j) && j \in [1, n-1] \\ \mathrm{LSM}_k(x)[j] &= \mathrm{lce}_k(x^R, 0, n-1-j) && j \in [0, n-2]. \end{aligned}$$

Using the method for answering lce_k queries, we can calculate a single value of LPM or LSM in $\mathcal{O}(k)$ time, implying a total time of $\mathcal{O}(kn)$ required to fully calculate both arrays. In fact the complexity of the full algorithm is bounded by this procedure.

A further set of identities allows us to compute the LPM_{k+1} and LSM_{k+1} arrays from the LPM_k and LSM_k arrays in $\mathcal{O}(1)$ time per entry, such that the arrays are progressively constructed, with each intermediate step yielding valuable information:

$$\begin{aligned} \mathrm{LPM}_{k+1}(x)[j] &= p + 1 + \mathrm{lce}(x, p+1, j+p+1) \\ \mathrm{LSM}_{k+1}(x)[j] &= s + 1 + \mathrm{lce}(x^R, s+1, n-j+s) \end{aligned}$$

$$\text{where } p = \mathrm{LPM}_k(x)[j] \text{ and } s = \mathrm{LSM}_k(x)[n-1-j].$$

After computing $\mathrm{LPM}_{k'}$ and $\mathrm{LSM}_{k'}$, for $0 \leq k' \leq k$, we may determine if a given string x of length $n \geq 2$ is a k-closed string by checking against three conditions for each k', as shown by Lem. 7.16. Recall that in the case when $n = 0$ or $n = 1$, x is trivially k-closed by definition.

Lemma 7.16. *Given a string x of length $n \geq 2$ and a natural number k, $0 \leq k < n$, x is k-closed if and only if there exists some $j \in \{1, \ldots, n-1\}$ and some $k' \in \{0, \ldots, k\}$ such that all the following conditions hold:*

(1) $j + \mathrm{LPM}_{k'}(x)[j] = n$

(2) $\forall i < j,\ \mathrm{LPM}_{k'}(x)[i] < \mathrm{LPM}_{k'}(x)[j]$

(3) $\forall i > n-1-j,\ \mathrm{LSM}_{k'}(x)[i] < \mathrm{LSM}_{k'}(x)[n-1-j]$.

Proof. Recall that $n \geq 2$. The three conditions can be seen to be necessary and sufficient for a string to be k-closed by considering the cases individually.

($\Longrightarrow$) Suppose Conditions 1-3 hold. We need to show that x is k-closed. We first prove that the conditions imply x is k'-pseudo-closed. In other words, we need to find a prefix u of x and a suffix v of x such that:

(I) $u \equiv_{k'} v$

(II) Except for u and v, there exists no length-$|u|$ factor w of x such that $w \equiv_{k'} u$ or $w \equiv_{k'} v$.

First, Condition 1 implies that the longest prefix match within k' errors starting at j terminates at position $n-1$ in x. This implies that $u = x[0\,..\,n-1-j] \equiv_{k'} x[j\,..\,n-1] = v$. Hence, (I) is true.

By contrary of (II), we have either (1) a factor w starting at position $i < j$ such that $w \equiv_{k'} u$ or (2) a factor w ending at position $i > n-1-j$ such that $w \equiv_{k'} v$.

For (1), this means that $\mathrm{LPM}_{k'}(x)[i] \geq \mathrm{LPM}_{k'}(x)[j]$. However, this contradicts Condition 2.

For (2), this means that $\mathrm{LSM}_{k'}(x)[i] \geq \mathrm{LSM}_{k'}(x)[n-1-j]$. However, this contradicts Condition 3.

Hence, both (I) and (II) are true. This implies that x is k'-pseudo-closed. Since $0 \leq k' \leq k$, we may further imply by Lem. 7.5 that x is k-closed.

($\Longleftarrow$) If x is k-closed, there must exist some k', where $0 \leq k' \leq k$, such that x is k'-pseudo-closed, by Lem. 7.5. For such a k', there is an associated k'-pseudo-closed-border consisting of some proper prefix u and some proper suffix v with equal length, such that $\delta_H(u,v) \leq k'$. We denote j where $v = x[j\,..\,n-1]$ and consequently $u = x[0\,..\,n-1-j]$. The longest prefix match $\mathrm{LPM}_{k'}(x)[j]$ starting at j must be greater than or equal to $|u|$ as $u \equiv_{k'} v$, yet it may not exceed the bounds of x and is therefore less than or equal to $|v|$. Therefore $\mathrm{LPM}_{k'}(x)[j] = |u| = |v| = n-j \implies \mathrm{LPM}_{k'}(x)[j] + j = n$ which implies Condition 1. From the definition of k-closed strings we also conclude that there exists no factor w of x with length $|w| = |u| = |v|$ such that $\delta_H(u,w) \leq k'$ or $\delta_H(v,w) \leq k'$. Therefore if we choose $i < j$ it must be the case that $\mathrm{LPM}_{k'}(x)[i] < \mathrm{LPM}_{k'}(x)[j]$, since otherwise we would have a $w \equiv_{k'} v$ starting at i which cannot be the case, and therefore we conclude Condition 2. Similarly if we choose $i > n-1-j$ it must be the case that $\mathrm{LSM}_{k'}(x)[i] < \mathrm{LSM}_{k'}(x)[n-1-j]$, since otherwise we would have a $w \equiv_{k'} v$ ending at i which cannot be the case, and therefore we conclude Condition 3. Thus all three conditions are satisfied. □

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$x[j]$	A	T	T	A	T	A	A	T	A	T	A	A	T	A	T
$\mathrm{LPM}_2[j]$	-1	3	4	7	2	10	4	4	7	2	5	4	3	2	1
$\mathrm{LSM}_2[j]$	1	2	3	4	5	2	7	6	2	10	2	5	7	2	-1
Cond. 1	F	F	F	F	F	T	F	F	T	F	T	T	T	T	T
Cond. 2	T	T	T	T	F	T	F	F	F	F	F	F	F	F	F
Cond. 3	T	T	T	F	F	T	F	F	F	F	F	F	F	F	F
	F	F	F	F	F	T	F	F	F	F	F	F	F	F	F

▲

Cond. 1 $j + \mathrm{LPM}_2[j] = n$

Cond. 2 $\mathrm{LPM}_2\_\mathrm{peaks}[j]$

Cond. 3 $\mathrm{LSM}_2\_\mathrm{peaks}[n-1-j]$

FIGURE 7.17: 2-closed border of length $n-j=10$ found at $j=5$ for string x of length $n=15$. This corresponds to the border strings ATTATAATAT and AATATAATAT which are at Hamming distance 1 from each other.

7.3.3 Main result

Theorem 7.1. *Given a string x of length n over an integer alphabet and a natural number k, $0 < k < n$, the k-closed border of x, if it exists, can be determined in $\mathcal{O}(kn)$ time and $\mathcal{O}(n)$ space.*

Proof. By Lem. 7.16, the time taken to determine whether a string x of length n is k-closed (and determine the k-closed border itself) is bounded by the computation of the $\mathrm{LPM}_{k'}(x)$ and $\mathrm{LSM}_{k'}(x)$ arrays, for all $0 \leq k' \leq k$. For a single k', Condition 1 trivially requires $\mathcal{O}(n)$ time to check across all possible j. Condition 2 and Condition 3 can be answered for each j in $\mathcal{O}(1)$ time by first preprocessing the $\mathrm{LPM}_{k'}(x)$ and $\mathrm{LSM}_{k'}(x)$ arrays in $\mathcal{O}(n)$ time to determine where the appropriate peaks lie (inspect Fig. 7.17 for an example). Therefore a total of $\mathcal{O}(kn)$ time is required to check across all possible k' and j as shown in Lem. 7.16. The $\mathrm{LPM}_{k'}(x)$ and $\mathrm{LSM}_{k'}(x)$ arrays can be updated for one k' value to the next one and so the space required is therefore only $\mathcal{O}(n)$. □

7.4 Pseudo-code

A full implementation of the algorithm was produced and the resulting pseudocode is presented here. The entry point of the algorithm is GETBORDER. This accepts a string

x of length n in addition to a parameter k specifying the maximum number of errors. The length that determines the k-closed border is returned. Note that if the string x is not k-closed, the function returns -1. We also make use of these additional functions:

reverse(x)

Standard library function. Accepts a string or array x of length n and returns the reversed string or array, respectively.

LCE(x, i, j)

Function to calculate longest common extension. Given a solid string x returns the length of the longest common prefix between the ith and jth suffix of x. Open source implementations are available using suffix trees or suffix arrays [115].

Function GETBORDER(x, n, k)

input :

x = Text string.

n = Length of text string x.

k = Specifies a k-closed query.

output:

Length of the k-closed border of x if it exists, or -1 otherwise.

if $n == 0$ **or** $n == 1$ **then**
    return 0

lpm $\leftarrow$ integer array of length n filled with -1

lsm $\leftarrow$ integer array of length n filled with -1

for $i \leftarrow 0$ **to** k **do**

    lpm = GETNEXTLPM(lpm, x, n)

    lsm = REVERSE(GETNEXTLPM(REVERSE(lpm), REVERSE(x), n))

    lpm_peaks $\leftarrow$ GETPEAKS(lpm)

    lsm_peaks $\leftarrow$ GETPEAKS(lsm)

    for $j \leftarrow 1$ **to** $n - 1$ **do**
        if $j +$ lpm$[j] == n$ **and** lpm_peaks$[j]$ **and** lsm_peaks$[n - 1 - j]$ **then**
            return $n - j$

return -1

Function GETNEXTLPM(lpm, x, n)

input :

lpm = Integer array specifying current lpm value.

x = Text string.

n = Length of text string x.

output:

Integer array specifying the subsequent lpm value.

for $i \leftarrow 1$ **to** $n - 1$ **do**

  if lpm$[i] == n - 1$ **then**

    continue

  lpm$[i]$ = lpm$[i]$ + LCE$(x, \text{lpm}[i] + 1, i + \text{lpm}[i] + 1) + 1$

return lpm

Function GETPEAKS(A)

input :

A = Array of integer values.

output:

Array of booleans identifying peaks within A.

peaks $\leftarrow$ boolean array with same length as A

max_val $\leftarrow -1$

for $i \leftarrow 0$ **to** $n - 1$ **do**

  if values$[i] >$ max_val **then**

    peaks$[i]$ = True

    max_val = values$[i]$

  else

    peaks$[i]$ = False

return peaks

7.5 Performance Analysis

We implemented the algorithm as a script, which would decide whether a given string is indeed a closed string within k Hamming distance errors. The implementation was then tested over numerous input sequences taken from real DNA data.

For proof-of-concept experimentation, we made use of the `python` programming language under a GNU/Linux operating system [116]. All experiments were conducted on a Desktop PC using one core of Intel Core CPU i5-6600K at 3.50GHz.

Within the experiments, the task was to establish whether the elapsed time of the implemented algorithm does indeed grow linearly with n and linearly with k. Input data for the experiments was taken from extracts of Chromosome 1 of the human reference genome GRCh37 [117]. The value of k and n was varied exponentially, and the total time taken recorded. The standard convention of 1 DNA letter occupying 1 byte of space is used [118].

In addition, it was found during experimentation that although an $\mathcal{O}(n)$-sized data structure for RMQs (see Section 2.5) was being used to answer lce queries, it was slow in practice. Therefore, further experimentation was carried out using an $\mathcal{O}(n \log n)$-sized data structure for RMQs [119], which was indeed faster in practice (see also [120] in this regard). In Table 7.1, we see a guide for the conducted experiments. The result of the experiments presented in Fig. 7.2, Fig. 7.3 and Fig. 7.4 confirm fully the theoretical findings: the elapsed time of the implemented algorithm grows linearly with n and linearly with k.

	n vs. run-time	k vs. run-time
$\mathcal{O}(n)$ RMQs	7.2	7.4
$\mathcal{O}(n \log n)$ RMQs	7.3	

TABLE 7.1: Guide for experimental figures.

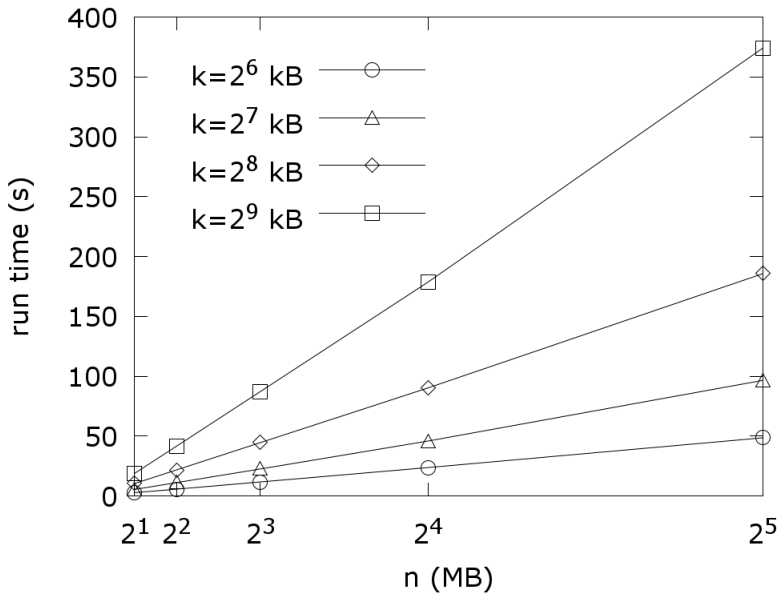

FIGURE 7.2: n vs. run-time with $\mathcal{O}(n)$-sized RMQs data structure.

In Fig. 7.2 we clearly see that the run-time increases linearly as the value of n increases. Additionally, we see the influence of k on the slope of the relationship, which indicates a linear correspondence between k and the run-time.

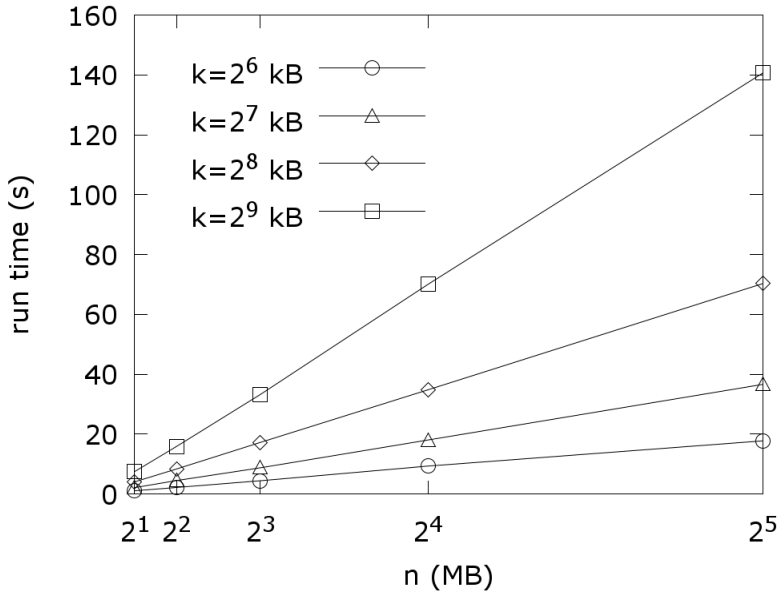

FIGURE 7.3: n vs. run-time with $\mathcal{O}(n \log n)$-sized RMQs data structure.

In Fig. 7.3 we see a formation of plots almost identical to those in Fig. 7.2, with the primary difference being the final run-time in all instances. The version of the algorithm using $\mathcal{O}(n \log n)$-sized RMQs appears to reduce the run-time without affecting the linear relationship, at least for the particular range of n and k values that were tested.

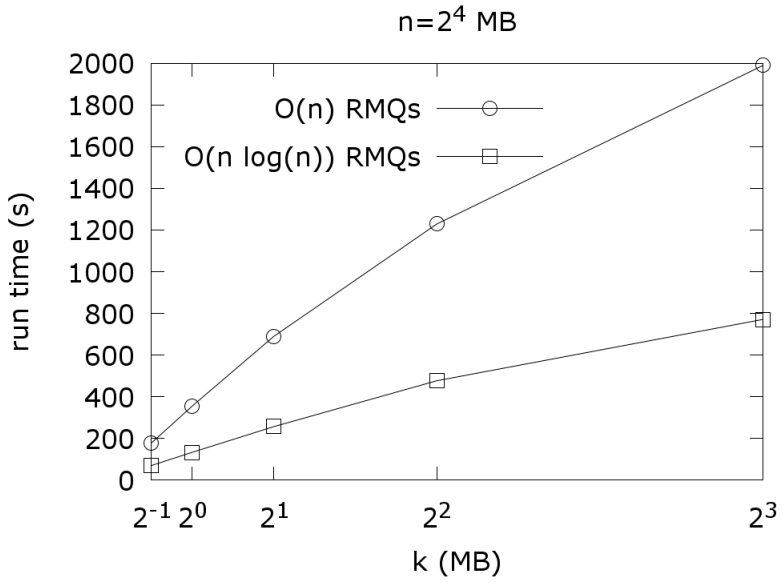

FIGURE 7.4: k vs. run-time with $\mathcal{O}(n)$- and $\mathcal{O}(n \log n)$-sized RMQs data structures

In Fig. 7.4 we may more accurately view the run-time comparison of the algorithm implemented with either $\mathcal{O}(n)$-sized or $\mathcal{O}(n \log n)$-sized RMQs. This illustrates the fact that the theoretical run-time may differ in practice, and a theoretically less efficient solution may be more practical for input values at lower orders of magnitude.

7.6 Conclusion

An algorithm has been presented in addition to proof-of-concept experiments for finding the k-closed border of a given string x of length n over an integer alphabet within Hamming distance k. The proposed algorithm was dependent on building two simple data structures, namely, $\mathrm{LPM}_k(x)$ and $\mathrm{LSM}_k(x)$. Given these data structures, it takes a further $\mathcal{O}(n)$ time to determine the k-closed border. The required space is $\mathcal{O}(n)$.

The main improvement could therefore be in the construction of these two tables, currently requiring $\mathcal{O}(kn)$ time. Decreasing this time complexity appears to be a reasonable, however non-trivial, goal for any future work on this problem, as any faster computation of $\mathrm{LPM}_k(x)$ and $\mathrm{LSM}_k(x)$ would imply a major breakthrough in approximate string matching under the Hamming distance model.

Chapter 8

Previous Factors

The longest previous factor of a string at a given position, provides the length of the longest factor or substring occurring at that location, which has previously occurred to the left of that position [121]. This chapter addresses the problem of creating a longest previous factor (LPF) array for degenerate strings, which is analogous to the well studied LPF array for solid strings.

Specifically, this chapter attempts to formalise an algorithm that may generate the LPF array for degenerate strings efficiently. This work is presented as the final research chapter of this thesis as it represents a work in progress, and therefore the format of this chapter differs slightly from prior chapters. Though a robust solution was not yet obtained for this particular problem, some improvements are noted here, which certainly show potential for further study.

8.1 Background

The most commonly cited application of identifying longest previous factors or the associated array, relates to the implementation of the Lempel-Ziv factorisation (also known as LZ77) [122, 123]. The LZ77 algorithm is one of a popular set of lossless data compression algorithms, that may be used for reducing the storage space of large datasets [124–126]. The algorithm compresses a given string by identifying repeated occurrences of factors and inserting a common placeholder which references a previous occurrence of such a repeated factor.

In this way, the identification of longest previous factors satisfies the condition that such factors are repeated and that they additionally represent a large quantity of data, by selecting for the longest such repeat.

The LZ77 factorisation permits more powerful compression than the alternative LZ78 factorisation, however the LZ77 is far more difficult to calculate. Fortunately, by knowing the longest previous factor array of a string, we may directly calculate the LZ77 factorisation [127].

Thus the ability to generate the longest previous factor array for degenerate strings, may permit the same efficiency in compression of degenerate strings. In the case of DNA alphabets, it remains to be seen if this would represent an improvement over simply substituting degenerate symbols with solid characters prior to compression, such as through the use of the IUPAC-encoding described in Section 6.

8.2 Problem Outline

8.2.1 Terminology

Much of the necessary terminology relating to degenerate strings has been covered in previous chapters. The definition of a degenerate string is outlined in Section 5.2.1. The definition of the solid equivalent of a degenerate string and associated matching table is also outlined in Section 5.3.1. Following from these definitions, some further terminology is introduced.

Definition 8.1. Given a degenerate string $\tilde{x}$, its solid equivalent $x_{\$}$ and extended matching table $\mathcal{M}$, consider a mismatch between two characters $x_{\$}[i]$ and $x_{\$}[j]$ where $x_{\$}[i] \neq x_{\$}[j]$. The mismatch is considered *real* (denoted $x_{\$}[i] \neq_{\text{real}} x_{\$}[j]$) or *fake* (denoted $x_{\$}[i] \neq_{\text{fake}} x_{\$}[j]$) accordingly:

$$x_{\$}[i] \neq_{\text{real}} x_{\$}[j] \iff x_{\$}[i] \neq x_{\$}[j] \text{ and } \mathcal{M}(x_{\$}[i], x_{\$}[j]) = \texttt{false}$$
$$x_{\$}[i] \neq_{\text{fake}} x_{\$}[j] \iff x_{\$}[i] \neq x_{\$}[j] \text{ and } \mathcal{M}(x_{\$}[i], x_{\$}[j]) = \texttt{true}$$

We may now define an lce query on $\tilde{x}$ in terms of a series of lce_k queries on $x_{\$}$.

Lemma 8.2. *Given a degenerate string $\tilde{x}$ of length n with k non-solid symbols, we may efficiently perform a single lce query on $\tilde{x}$ as follows:*

$$\text{lce}(\tilde{x}, i, j) = \min\{l \in S : x_{\$}[i + l] \neq_{real} x_{\$}[j + l]\}$$

$$\textit{where } S = \{\text{lce}_{k'}(x_{\$}, i, j) : \textit{lce}$$

This query requires at most $\mathcal{O}(k)$ time, after $\mathcal{O}(n)$ preprocessing of $\tilde{x}$.

Proof. Since $\tilde{x}$ and $x_{\$}$ are at Hamming distance k (differing at most k locations), it must be the case that $\text{lce}(\tilde{x}, i, j) = \text{lce}_{k'}(x_{\$}, i, j)$ for some $k' \in [0, k]$. Furthermore, since the lce is always terminated by a real mismatch, we have that $\text{lce}(\tilde{x}, i, j) = l \implies x_{\$}[i+l] \neq_{\text{real}} x_{\$}[j+l]$, providing an additional necessary condition. Let us assume that $\text{lce}(x_{\$}, i, j) = l$ which implies $\tilde{x}[i\,..\,i+l-1] \approx \tilde{x}[j\,..\,j+l-1]$, where l is not minimised. Therefore there exists an $l' < l$ where $x_{\$}[i+l'] \neq_{\text{real}} x_{\$}[j+l']$. However this is a contradiction, as $\text{lce}(\tilde{x}, i, j) = l \implies \tilde{x}[i+l'] \approx \tilde{x}[j+l']$. Therefore we must chose the minimum l satisfying the necessary conditions.

To determine the complexity of an $\text{lce}(\tilde{x}, i, j)$ query, we note that the query time is bounded by the calculation of the $k+1$ possible values $\text{lce}_{k'}(x_{\$}, i, j)$ for $0 \leq k' \leq k$. By calculating these in a recursive manner (kangaroo method [107, 108]), we may calculate all such values in $\mathcal{O}(k)$ time. Checking the extended matching table requires $\mathcal{O}(1)$ time for each k'. During the recursive calculation, the minimum k' satisfying the conditions is simply the first such satisfying k' encountered when calculating from $k' = 0$ up to $k' = k$. Therefore the time taken to process a single $\text{lce}(\tilde{x}, i, j)$ query is bounded by $\mathcal{O}(k)$. □

The main problem of this chapter relates to an extension of the longest previous factor problem, which is ordinarily applied to solid strings. Therefore, the extension of this definition as it relates to degenerate strings is shown in Def. 8.3.

Definition 8.3. The *longest previous factor array* (LPF) of a degenerate string $\tilde{x}$ of length n is an array storing the length of the longest factor at each position i in $\tilde{x}$ which matches a factor occurring to the left of i. Formally stated:

$$\text{LPF}_{\tilde{x}} = \max\{l : \tilde{x}[i\,..\,i+l-1] \approx \tilde{x}[j\,..\,j+l-1], 0 \leq j < i\}$$

8.2.2 Problem Statement

We now formally define the problem addressed within this chapter, namely calculating the Longest Previous Factor Array of a Degenerate String:

LONGEST PREVIOUS FACTOR ARRAY OF A DEGENERATE STRING

Input:
A degenerate string $\tilde{x}$ of length n.

Output:
The longest previous factor array $\text{LPF}_{\tilde{x}}$ of length n where $\text{LPF}_{\tilde{x}}[i]$ stores the length of the longest factor occurring at i within $\tilde{x}$ which matches some factor occurring to the left of i within $\tilde{x}$.

8.3 Naive Solution

As an initial baseline, the naive solution which utilises a brute-force approach is briefly covered here. This method arrives at a solution, with no particular regard for the space and time efficiency.

Given a degenerate string $\tilde{x}$ of length n with k non-solid symbols, we wish to determine the longest previous factor array of $\tilde{x}$, namely $\text{LPF}_{\tilde{x}}$. By Def. 8.3 we know that $\text{LPF}_{\tilde{x}}[i] = \max\{l : \tilde{x}[i\,..\,i+l-1] \approx \tilde{x}[j\,..\,j+l-1], 0 \leq j < i\} = \max\{\text{lce}(\tilde{x}, i, j) : 0 \leq j < i\}$. Therefore, by an exhaustive checking of all possibilities, a single instance of i requires at most n $\text{lce}_k(x_{\$}, i, j)$ queries, giving a time bound of $\mathcal{O}(kn)$. Therefore to determine the full $\text{LPF}_{\tilde{x}}$ array for $0 \leq i < n$ requires at most $\mathcal{O}(kn^2)$ time. Thus we arrive at a strict upper bound on execution time for the optimal solution.

8.4 Improved Solution

The aim is now to improve on the naive solution as described in Section 8.3. These improvements are detailed in 2 sections, namely Section 8.4.1 and Section 8.4.2. Each successive improvement from the naive solution is achieved by utilising an increasing number of data structures in the interest of reducing total algorithmic time complexity, at the expense of increasing space complexity. It should be noted here again, that neither of these improvements represents what could be deemed a final efficient solution, rather they are an initial attempt to solve the stated problem in Section 8.2, with the aim of further work in this area potentially providing a more robust solution.

8.4.1 Primary Improvement

To initially improve on the naive solution in Section 8.3, we perform some basic preprocessing. As an initial preprocessing step, the conservative degenerate text $\tilde{x}$ is transformed into its solid equivalent $x_{\$}$ by replacing each of the k non-solid symbols with unique symbols $\$_d$, where $0 \leq d < k \leq n$, as set out in Section 5.3.1.

We then calculate the longest previous factor array $\mathrm{LPF}_{x_{\$}}$ of the solid string $x_{\$}$ in $\mathcal{O}(n)$ time, via standard methods on solid strings. This will allow us to make inferences on the values of $\mathrm{LPF}_{\tilde{x}}$. Before proceeding, it is necessary to define some intervals of $x_{\$}$ that will prove useful.

We consider the string $x_{\$}$ to be divided into $k+1$ intervals, with consecutive intervals separated by a single character, with each such character at the location of one of the k formerly non-solid symbols. We describe these intervals as *seeds*, and refer to them with the notation $S_0, S_1, \ldots, S_k$. Any given pair of consecutive seeds S_i and S_{i+1} will be separated by the single unique character $\$_i$. This is illustrated in Fig. 8.4.

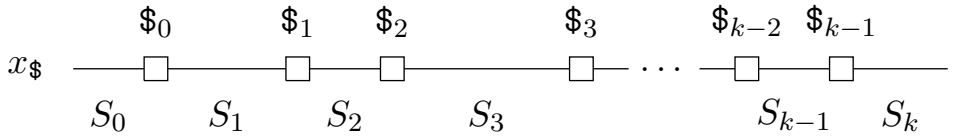

FIGURE 8.4: Example of $x_{\$}$ separated into seeds.

We use the function $\mathtt{pre}(d, l)$ to refer to the length l interval immediately *preceding* the unique character $\$_d$ within $x_{\$}$, with an additional constraint that the interval may not extend to the left in such a way that it includes any other symbol $\$_{d'} \neq \$_d$. The union of all k such intervals is denoted by $\mathtt{pre}(l)$. We present an example of these in Fig. 8.6

Definition 8.5. For a given solid equivalent $x_{\$}$ of a degenerate string $\tilde{x}$ of length n with k non-solid symbols, we define the following intervals:

$$\mathtt{pre}(d, l) = [\max\{\mathtt{loc}_{x_{\$}}(\$_d) - l, \mathtt{loc}_{x_{\$}}(\$_{d-1}) + 1\}, \mathtt{loc}_{x_{\$}}(\$_d)]$$
$$\mathtt{pre}(l) = \bigcup_{d=0}^{k-1} \mathtt{pre}(d, l)$$

Note that although there is no such character $\$_{-1}$ in the alphabet of $x_{\$}$, we assign $\mathtt{loc}_{x_{\$}}(\$_{-1}) = -1$ for the purpose of this definition.

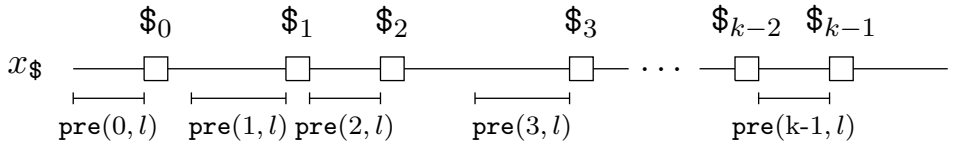

FIGURE 8.6: Example of $\texttt{pre}(d, l)$ intervals of $x_\$$ for some l.

It naturally follows from Def. 8.5, that $\texttt{pre}(d, l) \leq l$ for all d, and the factor of $x_\$$ described by the interval $\texttt{pre}(d, l)$ will never contain any symbol $\$_d$.

We now define an additional set of indexes within $x_\$$ that will prove useful. Given an index i within $x_\$$, we use the notation $\mathcal{L}_{x_\$}(i)$ to refer to the set of indexes to the left of i at which the longest previous factor at i occurs.

Definition 8.7. For a given solid equivalent $x_\$$ of a degenerate string $\tilde{x}$ we define the set of indexes $\mathcal{L}_{x_\$}(i)$ as follows:

$$\mathcal{L}_{x_\$}(i) = \{j < i : x_\$[j\,..\,j + \text{LPF}_{x_\$}[i] - 1] = x_\$[i\,..\,i + \text{LPF}_{x_\$}[i] - 1]\}$$

We now introduce the first crucial lemma within this section, that provides the basis of the various shortcuts taken by the algorithm to determine longest previous factors in degenerate strings, namely Lem. 8.8.

Lemma 8.8. *Given a degenerate string $\tilde{x}$ and its solid equivalent $x_\$$, we have that at every index i:*

$$\text{LPF}_{x_\$}[i] = l \implies \text{LPF}_{\tilde{x}}[i] \geq l$$

Lem. 8.8 gives us some indication of the value of $\text{LPF}_{\tilde{x}}[i]$ based on $\text{LPF}_{x_\$}[i]$. We now consider the character which follows the matched characters within a given $\text{LPF}_{x_\$}[i]$, namely $x_\$[i + l]$ where $l = \text{LPF}_{x_\$}[i]$. Based on the value of this character, we determine Def. 8.9, which describes results that follow from two mutually exclusive and exhaustive cases, dubbed Type I and Type II for a given i, where i is not the location of any character $\$_d$.

Definition 8.9. Given a degenerate string $\tilde{x}$ with k non-solid symbols and its solid equivalent $x_\$$ of length n, consider a position i in $x_\$$ (where $x_\$[i] \neq \$_d\ \forall d$) and its longest previous factor $\text{LPF}_{x_\$}[i] = l$. Based on the value of the character $x_\$[i + l]$, we characterise the index i as falling into one of the 2 following Types:

Type I $x_{\$}[i+l] \neq \$_d$, $\forall d$
$\implies \text{LPF}_{\tilde{x}}[i] = \max(\{\text{lce}(\tilde{x}, i, j) : j \in \bigcup_{d \in A} \texttt{pre}(d, l)\} \cup \{l\})$

Type II $\exists d$, $x_{\$}[i+l] = \$_d$ *or* $i + l = n$
$\implies \text{LPF}_{\tilde{x}}[i] = \max(\{\text{lce}(\tilde{x}, i, j) : j \in \mathcal{L}_{x_{\$}}(i) \cup \bigcup_{d \in A} \texttt{pre}(d, l)\} \cup \{l\})$

$$A = \{d : \texttt{loc}_{x_{\$}}(\$_d) < i\}$$

We use the notation $I_{x_{\$}}$ and $II_{x_{\$}}$ to refer to two mutually exclusive sets of all indexes i in $x_{\$}$ (where $x_{\$}[i] \neq \$_d \; \forall d$) which are characterised as Type I and Type II respectively.

When considering the indexes within a single seed from left to right, the Type I and Type II indexes comprise two distinct sets of consecutive indexes, effectively partitioning every seed into a left-hand Type I interval and a right-hand Type II interval. We state this formally in Lem. 8.10 and illustrate an example in Fig. 8.11.

Lemma 8.10. *Given a degenerate string $\tilde{x}$ and its solid equivalent $x_{\$}$, the following holds true for any given seed $S = x_{\$}[i\,..\,j]$:*

$$\exists! c \in [i, j+1] \text{ such that } p \in I_{x_{\$}} \; \forall p \in [i, c-1] \text{ and } p \in II_{x_{\$}} \; \forall p \in [c, j]$$

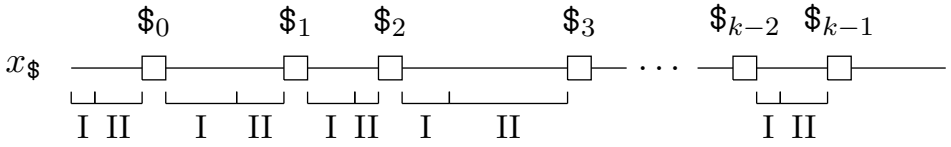

FIGURE 8.11: Example of Type I and Type II partitioning of seeds.

Def. 8.9 effectively provides a list of candidates at which the desired $\text{LPF}_{\tilde{x}}[i]$ can be found, based on the information given by $\text{LPF}_{x_{\$}}[i]$. Each candidate location j is considered, and the longest common extension between i and j is determined. The maximum longest common extension across all j values in the candidate set (as determined by the Type) provides the value of $\text{LPF}_{\tilde{x}}[i]$.

Considering the time complexity required to determine the full $\text{LPF}_{\tilde{x}}$ array for $0 \leq i < n$ following this primary improvement, we see that in the worst case, we still have a bound of $\mathcal{O}(kn^2)$ time. Though this is the theoretical worst case, in practice it appears that the improvements would tend to lead to a reduced complexity time, which can be seen when we consider the complexity in further detail.

The initial preprocessing required to build the required data structures takes $\mathcal{O}(n)$ time. Let us now consider the case for a single index i within $\tilde{x}$. If this index i is Type I, we have $\mathcal{O}(kl)$ candidates to be checked where $l = \text{LPF}_{x_\$}[i]$. Checking each of these candidates requires k time by the kangaroo method [107, 108], giving a total time complexity for this particular index i of $\mathcal{O}(k^2 l)$. Alternatively, if this index i is Type II, we have $\mathcal{O}(kl + |\mathcal{L}_{x_\$}(i)|)$ candidates to be checked, giving a total time complexity for this particular index i of $\mathcal{O}(k^2 l + k|\mathcal{L}_{x_\$}(i)|)$. Finally, in the case where i marks the location of some $\$_d$ in $x_\$$, we use the standard naive method of comparing i to all possible earlier indexes, requiring $\mathcal{O}(kn)$.

Thus the final time complexity across all i in practice is an amalgamation of various time complexities which are highly dependent on the structure of the underlying text, and may or may not approach $\mathcal{O}(kn^2)$ time, though it can be guaranteed to be theoretically no worse than this.

8.4.2 Secondary Improvement

Further improvements may be made by building on the solution as described in 8.4.1. This will require several additional data structures which will be further outlined here.

The first of these is the *alphabet occurrence list* of a text, which stores the locations of each character in the alphabet of the text. We define this formally in Def. 8.12.

Definition 8.12. Given a text x of length n over the alphabet Σ we define the alphabet occurrence list $\mathcal{A}_x$ as a function mapping every character $\sigma \in \Sigma$ to a subset of the set of integers $\{0, \ldots, n-1\}$ corresponding to the index locations of σ in x:

$$\mathcal{A}_x : \Sigma \to \{0, \ldots, n-1\}$$

$$\mathcal{A}_x(\sigma) = \{i : x[i] = \sigma\}$$

It is clear that the alphabet occurrence list may be constructed in $\mathcal{O}(n)$ time and space, with a single pass of the text x of length n.

Next we define the *tail matrix* of a degenerate string $\tilde{x}$, which stores a subset of all the possible lce queries within $\tilde{x}$ (taking into account degenerate matches), choosing only those queries that include a $\$_d$ character. We define this formally in Def. 8.13.

Definition 8.13. Given a degenerate string $\tilde{x}$ of length n over the alphabet Σ with k non-solid symbols and its solid equivalent $x_\$$, we define the $k \times n$ tail matrix $\mathcal{B}$ as follows:

$$\mathcal{B}[i][j] = \text{lce}(\tilde{x}, \texttt{loc}_{x_\$}(\$_i), j) \quad 0 \leq i < k \quad 0 \leq j < n$$

The tail matrix requires significant space $\mathcal{O}(kn)$, however this in turn provides a significant time saving for the algorithm after preprocessing. The time taken to generate the tail matrix naively is $\mathcal{O}(k^2 n)$, though this can be reduced to $\mathcal{O}(kn)$ time by reusing previously calculated values in the matrix.

Finally, in addition to these data structures, we will make use of the reverse of $x_\$$ which we refer to as $x_\$^R$. We preprocess $x_\$^R$ in $\mathcal{O}(n)$ time by standard methods, to enable constant time lce queries on $x_\$^R$.

For each index i in the text that does not correspond to some $\$_d$, we define the values $c_I(\text{i})$ and $c_{II}(\text{i})$ which serve as candidates for the value of $\text{LPF}_{\tilde{x}}[i]$, in addition to the candidate $l = \text{LPF}_{x_\$}[i]$. The definitions of $c_I(\text{i})$ and $c_{II}(\text{i})$ follow naturally from Def. 8.9 and form the foundation of Lem. 8.15, which enables us to consider the calculation of the c_I and c_{II} values independently.

Definition 8.14. Given a degenerate string $\tilde{x}$ and its solid equivalent $x_\$$ we define the values $c_I(i)$ and $c_{II}(i)$ as follows:

$$c_I(i) = \max\{\text{lce}(\tilde{x}, i, j) : j \in \bigcup_{d \in A} \texttt{pre}(d, l)\}$$
$$c_{II}(i) = \max\{\text{lce}(\tilde{x}, i, j) : j \in \mathcal{L}_{x_\$}(i)\}$$

$$A = \{d : \texttt{loc}_{x_\$}(\$_d) < i\}$$

Lemma 8.15. *Given a degenerate string $\tilde{x}$ and its solid equivalent $x_\$$ we may determine the value $\text{LPF}_{\tilde{x}}[i]$ from $l = \text{LPF}_{x_\$}[i]$ and values $c_I(i)$, $c_{II}(i)$ as follows:*

$$\text{LPF}_{\tilde{x}}[i] = \max\{l, c_I(i)\} \qquad i \in I_{x_\$}$$
$$\text{LPF}_{\tilde{x}}[i] = \max\{l, c_I(i), c_{II}(i)\} \qquad i \in II_{x_\$}$$

Note that in Lem. 8.15 we only make use of $c_{II}(i)$ in the Type II scenario, whereas in the Type I scenario it need not be calculated. In the case of i where $x_\$[i] = \$_d$ for some d, we use the usual naive method for determining $\text{LPF}_{\tilde{x}}[i]$.

The algorithm proceeds by considering each i in the text and applying Lem. 8.15 accordingly. In the case of $c_I(\text{i})$, the method of calculation remains unchanged, whereas the calculation of $c_{II}(\text{i})$ is where a significant improvement may be found, which may now be described.

Given a degenerate string $\tilde{x}$ and its solid equivalent $x_{\$}$ and an index $i \in II_{x_{\$}}$, we wish to calculate $c_{II}(\mathrm{i})$. The method for a given seed S of $x_{\$}$ will be explained, which may then be applied to all other seeds of $x_{\$}$ in the same way. This seed S will have some unique index s which marks the boundary between Type I and Type II indexes, namely $s \in S$ and $s \in II_{x_{\$}}$ whereas $s-1 \notin II_{x_{\$}}$. Let us calculate $L = \mathrm{LPF}_{x_{\$}}[s]$ and create an array C of size L with all values initially set to 0.

Additionally, we know that the character at index $i+L$ immediately precedes some $\$_d$. Let us call this character $\alpha = x_{\$}[i+L]$ and its occurring index $p = i+L$. Let us also consider the set $\mathcal{A}_{x_{\$}}(\alpha)$ providing all occurrences of the character α, and define a subset $D = \{i < p : i \in \mathcal{A}_{x_{\$}}(\alpha)\}$ containing only those indexes corresponding to locations preceding p.

Having determined the index p and set D we now proceed to take the reverse lce on $x_{\$}$ at p and every occurrence of α, i.e. we calculate $l_j = \mathrm{lce}(x_{\$}^R, n-1-p, n-1-j)$ where $j \in D$. For each such l_j we append the appropriate tail according to the $\mathcal{B}$ tail matrix. Specifically, for each l_j we obtain $l'_j = l_j + \mathcal{B}[d][j+1]$ where d is such that the current seed $S = S_d$.

Whenever the value of a calculated l'_j is larger than $C[l_j]$, we set $C[l_j] = l'_j$. At the conclusion of this process, this enables C to provide us with the maximum possible l'_j for a given l_j across all occurrences of α to the left of p. Finally, we parse C from left to right, and set $C[i] = \max\{C[i], C[i-1]-1\}$ for $0 < i < L$.

The array C now provides us with the desired $c_{II}(i)$ values within this seed S. Specifically, for an index i within seed S, we have that $c_{II}(i) = C[i-s]$.

8.5 Conclusion

We conclude that the above improvements represent an initial look into solving the problem stated in Section 8.2.2. Unlike previous chapters, there is currently no formal outline of the final proposed algorithm or proof of concept. Instead this chapter provides an outline of a series of suggested improvements, which may act as a starting point for further research.

Thus the purpose of this chapter is not only a presentation of research carried out, but a motivator for any reader seeking to delve further into this particular problem. The topic of longest previous factors on degenerate strings is ripe for further research [128–130], and the ideas presented here should be seen as a springboard for those with the motivation to further pursue this problem.

Chapter 9

Conclusion

Throughout this thesis, multiple string structures have been explored: circular strings, abelian palindromes, maximal palindromes, inverted repeats, closed strings and previous factors. For each such structure, a problem or set of problems was formally defined, and an algorithmic solution presented, often alongside an analysis and implementation.

In each instance, varying degrees of improvement have been made in the space and time efficiency of solutions to each problem. Contrastingly, some of the presented problems have been more novel, with no pre-existing solution from which they were built upon.

The main contributions of each research chapter along with suggestions for further work, are summarised below:

Chapter 3: Circular Strings

Contributions

- Developed a set of filters to reduce the search space required when performing circular string searches.
- Produced a web interface for solving the Approximate Circular Pattern Matching (ACPM) problem.
- Created a client-side implementation reducing the need for the upload of large data sets.
- Analysed the run-time of the implementation and confirmed correctness of the output.
- Estimated the average reduction of search space enabled by the filters.
- Published the source code of the implementation online.

Further Work

- Improve the implementation to permit input of common DNA data files e.g. FASTA, FASTQ, EMBL.
- Develop additional filters with linear complexity, to supplement the current set of filters.
- Perform further analysis of the influence of the approximation level k on the algorithmic run-time.

Chapter 4: Abelian Palindromes

Contributions

- Defined the concept of an *abelian palindrome* which satisfies the property of being abelian equivalent to some palindrome of the same length.
- Defined a new data structure known as an *abelian palindromic array*, which can be used to determine if any factor of a string is abelian palindromic in $\mathcal{O}(1)$ time.
- Defined a new data structure known as a *prefix parity integer array*.
- Developed an algorithm identifying whether or not a string is abelian palindromic, requiring $\mathcal{O}(n)$ time for a string of length n with an alphabet size $|\Sigma| \leq \log_2(n)$.
- Developed an algorithm to build an abelian palindromic array for a string of length n, with an alphabet of size $|\Sigma| \leq \log_2(n)$ requiring $\mathcal{O}(n)$ preprocessing time and $\mathcal{O}(n + |\Sigma|)$ space.

Further Work

- Determine possible applications of identifying abelian palindromes efficiently.
- Modify the algorithm to remove the requirement that the alphabet $|\Sigma| \leq \log_2(n)$ for a string of length n.
- Consider alternative theoretical solutions making use of the bit manipulation paradigms within quantum computing.

Chapter 5: Maximal Palindromes

Contributions

- Developed an algorithm that is capable of producing a sequence of maximal palindromic factors corresponding to a given degenerate string of length n over an alphabet Σ with no more than k non-solid symbols, requiring $\mathcal{O}(k|\Sigma|(k + \log|\Sigma|) + kn)$ time and $\mathcal{O}(k(k + |\Sigma|) + n)$ space, along with the pseudocode of an implementation.

- Developed a technique to reduce the time and space complexity to $\mathcal{O}(n \log(n))$ and $\mathcal{O}(n)$ respectively, under the assumption that $k \leq \log(n)$ and $|\Sigma|$ is a small constant.
- Tested the efficacy of the algorithm, identifying average run-times and the probability of successfully identifying a factorisation for a random string.

Further Work

- Determine a method to remove the restriction that all palindromes in the factorisation are maximal.
- Extend the algorithm to permit small gaps between consecutive factors to improve the practical applications of the algorithm.

Chapter 6: Inverted Repeats

Contributions

- Defined an alternative complement matching system for IUPAC characters.
- Created a software implementation IUPACpal that is capable of identifying inverted repeats with gaps and possible mismatches, according to the complement matching scheme.
- Performed run-time analysis to determine that IUPACpal compares favourably to a similar application packaged with EMBOSS.
- Demonstrated that IUPACpal identifies previously unidentified inverted repeats when compared with EMBOSS.
- Created a command-line interface for IUPACpal that mimics the interface of EMBOSS.
- Made all source code of our implementation publicly available online.

Further Work

- Add additional functionality to IUPACpal which is already native to the EMBOSS tool, such as the ability to handle multiple DNA data file formats.
- Carry out further performance tests to compare IUPACpal to other commonly used software alternatives.

Chapter 7: Closed Strings

Contributions

- Defined an extension of k-closed strings in addition to 3 other related definitions.

- Defined two new data structures created from a given string x, namely $\mathrm{LPM}_k(x)$ and $\mathrm{LSM}_k(x)$.
- Developed an algorithm to determine if a given string of length n is k-closed, requiring $\mathcal{O}(kn)$ time and $\mathcal{O}(n)$ space.
- Carried out proof-of-concept experiments to confirm the correctness of the algorithm and determine average run-times.

Further Work

- Improve the algorithmic efficiency required to generate $\mathrm{LPM}_k(x)$ and $\mathrm{LSM}_k(x)$ for a given string x.
- Generalise other previously solved closed string problems as k-closed string problems using the newly provided definition.

Chapter 8: Previous Factors

Contributions

- Performed an initial exploration of a solution to the problem of finding the longest previous factors in a degenerate string.
- Described several data structures and tools that may be relevant to solving the problem.

Further Work

- As an newly defined problem, this area is ripe for more research and the specific problem as stated remains unsolved.

During the research period that preceded the writing of this thesis, there were additional side problems worked on and explored to varying degrees of success. From the outcomes that were obtained, the previous chapters represent the most fruitful. In the case where further avenues of research are open, we have attempted to highlight this within the listings of further work above.

In summary, this thesis presents an amalgamation of those projects which offered the most value as a body of research, and we believe this thesis provides a genuine contribution to the current body of knowledge in this field. We certainly invite the interested reader to further contribute to the work presented here, with the outlined further work above provided as a guide.

Bibliography

[1] Gordon E Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[2] Ethan Mollick. Establishing moore's law. *IEEE Annals of the History of Computing*, 28(3):62–75, 2006.

[3] Thomas N Theis and H-S Philip Wong. The end of moore's law: A new beginning for information technology. *Computing in Science & Engineering*, 19(2):41–50, 2017.

[4] James R Powell. The quantum limit to moore's law. *Proceedings of the IEEE*, 96 (8):1247–1248, 2008.

[5] Laszlo B Kish. End of moore's law: thermal (noise) death of integration in micro and nano electronics. *Physics Letters A*, 305(3-4):144–149, 2002.

[6] Vivien Marx. Biology: The big challenges of big data. *Nature*, 2013.

[7] Eve S McCulloch. Harnessing the power of big data in biological research. *BioScience*, 63(9):715–716, 2013.

[8] Yuichi Kodama, Martin Shumway, and Rasko Leinonen. The sequence read archive: explosive growth of sequencing data. *Nucleic acids research*, 40(D1): D54–D56, 2012.

[9] Jemal Abawajy. Comprehensive analysis of big data variety landscape. *International journal of parallel, emergent and distributed systems*, 30(1):5–14, 2015.

[10] Barna Saha and Divesh Srivastava. Data quality: The other face of big data. In *2014 IEEE 30th International Conference on Data Engineering*, pages 1294–1297. IEEE, 2014.

[11] Kerry G Coffman and Andrew M Odlyzko. *Internet growth: Is there a "Moore's Law" for data traffic?*, pages 47–93. Springer, 2002.

[12] Ibrahim Abaker Targio Hashem, Ibrar Yaqoob, Nor Badrul Anuar, Salimah Mokhtar, Abdullah Gani, and Samee Ullah Khan. The rise of "big data" on cloud computing: Review and open research issues. *Information systems*, 47:98–115, 2015.

[13] Elaine R Mardis. Dna sequencing technologies: 2006–2016. *Nature protocols*, 12 (2):213, 2017.

[14] Eric D Green, Edward M Rubin, and Maynard V Olson. The future of dna sequencing. *Nature News*, 550(7675):179, 2017.

[15] Jay Shendure and Erez Lieberman Aiden. The expanding scope of dna sequencing. *Nature biotechnology*, 30(11):1084, 2012.

[16] Swagath Venkataramani, Srimat T Chakradhar, Kaushik Roy, and Anand Raghunathan. Approximate computing and the quest for computing efficiency. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2015.

[17] Heidi L Rehm. Evolving health care through personal genomics. *Nature Reviews Genetics*, 18(4):259, 2017.

[18] James B Stewart and Patrick F Chinnery. The dynamics of mitochondrial dna heteroplasmy: implications for human health and disease. *Nature Reviews Genetics*, 16(9):530–542, 2015.

[19] Merly Escalona, Sara Rocha, and David Posada. A comparison of tools for the simulation of genomic next-generation sequencing data. *Nature Reviews Genetics*, 17(8):459, 2016.

[20] EMBOSS PALINDROME. emboss.sourceforge.net/apps/cvs/emboss/apps/palindrome.html, n.d. Accessed: 2020-06-16.

[21] Peter Rice, Ian Longden, and Alan Bleasby. Emboss: the european molecular biology open software suite, 2000.

[22] Maxime Crochemore and Wojciech Rytter. *Jewels of stringology: text algorithms*. World Scientific, 2002.

[23] Zvi Galil. Open problems in stringology. In *Combinatorial Algorithms on Words*, pages 1–8. Springer, 1985.

[24] Shan Muthukrishnan and Krishna Palem. Non-standard stringology: Algorithms and complexity. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 770–779, 1994.

[25] Simon Rubinstein-Salzedo. Big o notation and algorithm efficiency. In *Cryptography*, pages 75–83. Springer, 2018.

[26] Stephen A Cook. An overview of computational complexity. *Communications of the ACM*, 26(6):400–408, 1983.

[27] Ian Chivers and Jane Sleightholme. An introduction to algorithms and the big o notation. In *Introduction to Programming with Fortran*, pages 359–364. Springer, 2015.

[28] Huy N Nguyen and Krzysztof Onak. Constant-time approximation algorithms via local improvements. In *2008 49th Annual IEEE Symposium on Foundations of Computer Science*, pages 327–336. IEEE, 2008.

[29] Robert M Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development*, 11(1):25–33, 1967.

[30] Shmuel Winograd. *Arithmetic complexity of computations*, volume 33. Siam, 1980.

[31] Siamak Tazari and Matthias Müller-Hannemann. Dealing with large hidden constants: Engineering a planar steiner tree ptas. *Journal of Experimental Algorithmics (JEA)*, 16:3–1, 2008.

[32] Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007.

[33] Richard W Hamming. Error detecting and error correcting codes. *The Bell system technical journal*, 29(2):147–160, 1950.

[34] Maxime Crochemore. String-matching on ordered alphabets. *Theoretical Computer Science*, 92(1):33–47, 1992.

[35] Jean Pierre Duval. Factorizing words over an ordered alphabet. *Journal of Algorithms*, 4(4):363–381, 1983.

[36] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Annual Symposium on Combinatorial Pattern Matching*, pages 181–192. Springer, 2001.

[37] Béla Bollobás and Shoham Letzter. Longest common extension. *European Journal of Combinatorics*, 68:242–248, 2018.

[38] Robert Giegerich and Stefan Kurtz. From ukkonen to mccreight and weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*, 19(3):331–353, 1997.

[39] Peter Brass. *Advanced data structures*, volume 193. Cambridge University Press Cambridge, 2008.

[40] Rodrigo Cánovas and Gonzalo Navarro. Practical compressed suffix trees. In *International Symposium on Experimental Algorithms*, pages 94–105. Springer, 2010.

[41] Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.

[42] Martin Farach. Optimal suffix tree construction with large alphabets. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 137–143. IEEE, 1997.

[43] Dan Gusfield. Algorithms on stings, trees, and sequences: Computer science and computational biology. *Acm Sigact News*, 28(4):41–60, 1997.

[44] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

[45] Johannes Fischer and Volker Heun. A new succinct representation of rmq-information and improvements in the enhanced suffix array. In *International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, pages 459–470. Springer, 2007.

[46] Erik D Demaine, Gad M Landau, and Oren Weimann. On cartesian trees and range minimum queries. In *International Colloquium on Automata, Languages, and Programming*, pages 341–353. Springer, 2009.

[47] Niklas Baumstark, Simon Gog, Tobias Heuer, and Julian Labeit. Practical range minimum queries revisited. In *16th International Symposium on Experimental Algorithms (SEA 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[48] Carl Barton, Costas S Iliopoulos, and Solon P Pissis. Fast algorithms for approximate circular string matching. *Algorithms for Molecular Biology*, 9(1):9, 2014.

[49] Costas S Iliopoulos, Solon P Pissis, and M Sohel Rahman. Searching and indexing circular patterns. In *Algorithms for Next-Generation Sequencing Data*, pages 77–90. Springer, 2017.

[50] Md Aashikur Rahman Azim, Mohimenul Kabir, and M Sohel Rahman. A simple, fast, filter-based algorithm for circular sequence comparison. In *International Workshop on Algorithms and Computation*, pages 183–194. Springer, 2018.

[51] Gonzalo Navarro and Mathieu Raffinot. *Flexible pattern matching in strings: practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002.

[52] Ricardo A Baeza-Yates and Chris H Perleberg. Fast and practical approximate string matching. In *Annual Symposium on Combinatorial Pattern Matching*, pages 185–192. Springer, 1992.

[53] Gonzalo Navarro. A guided tour to approximate string matching. *ACM computing surveys (CSUR)*, 33(1):31–88, 2001.

[54] Horst Bunke and Urs Bühler. Applications of approximate string matching to 2d shape recognition. *Pattern recognition*, 26(12):1797–1812, 1993.

[55] Spencer Bliven and Andreas Prlić. Circular permutation in proteins. *PLoS computational biology*, 8(3), 2012.

[56] Wei-Cheng Lo, Chi-Ching Lee, Che-Yu Lee, and Ping-Chiang Lyu. Cpdb: a database of circular permutation in proteins. *Nucleic acids research*, 37(suppl_1): D328–D332, 2009.

[57] R Dulbecco and M Vogt. Evidence for a ring structure of polyoma virus dna. *Proceedings of the National Academy of Sciences of the United States of America*, 50(2):236, 1963.

[58] Roger Weil and Jerome Vinograd. The cyclic helix and cyclic coil forms of polyoma viral dna. *Proceedings of the National Academy of Sciences of the United States of America*, 50(4):730, 1963.

[59] Martin Thanbichler, Sherry C Wang, and Lucy Shapiro. The bacterial nucleoid: a highly organized and dynamic structure. *Journal of cellular biochemistry*, 96(3): 506–521, 2005.

[60] Abhishek Satishchandran and David B Weiner. Plasmids: current research and future trends. *Expert Review of Vaccines*, 8(1):33–34, 2009.

[61] Thorsten Allers and Moshe Mevarech. Archaeal genetics—the third way. *Nature Reviews Genetics*, 6(1):58–73, 2005.

[62] Francisco Fernandes, Luísa Pereira, and Ana T Freitas. Csa: an efficient algorithm to improve circular dna multiple alignment. *BMC bioinformatics*, 10(1):230, 2009.

[63] Axel Mosig, Ivo L Hofacker, and Peter F Stadler. Comparative analysis of cyclic sequences: viroids and other small circular rnas. In *German Conference on Bioinformatics*. Gesellschaft für Informatik eV, 2006.

[64] Taehyung Lee, Joong Chae Na, Heejin Park, Kunsoo Park, and Jeong Seop Sim. Finding optimal alignment and consensus of circular strings. In *Annual Symposium on Combinatorial Pattern Matching*, pages 310–322. Springer, 2010.

[65] Ricardo Betancur-R, Gavin JP Naylor, and Guillermo Ortí. Conserved genes, sampling error, and phylogenomic inference. *Systematic biology*, 63(2):257–262, 2014.

[66] Rohit J Parikh. On context-free languages. *Journal of the ACM (JACM)*, 13(4): 570–581, 1966.

[67] Donald E Knuth, James H Morris, Jr, and Vaughan R Pratt. Fast pattern matching in strings. *SIAM journal on computing*, 6(2):323–350, 1977.

[68] Etzard Stolte and Gustavo Alonso. Optimizing scientific databases for client side data processing. In *International Conference on Extending Database Technology*, pages 390–408. Springer, 2002.

[69] Random.org. https://www.random.org/clients/http/, n.d. Accessed: 2020-06-16.

[70] Bacem Saada and Jing Zhang. Dna sequences compression algorithm based on extended-ascii representation. In *Proceedings of the world congress on engineering and computer science*, volume 2, 2015.

[71] Elizabeth Pollom, Kristen K Dang, E Lake Potter, Robert J Gorelick, Christina L Burch, Kevin M Weeks, and Ronald Swanstrom. Comparison of siv and hiv-1 genomic rna structures reveals impact of sequence evolution on conserved and non-conserved structural motifs. *PLoS pathogens*, 9(4), 2013.

[72] Sandeep Subramanian, Srilakshmi Chaparala, Viji Avali, and Madhavi K Ganapathiraju. A pilot study on the prevalence of dna palindromes in breast cancer genomes. *BMC medical genomics*, 9(3):73, 2016.

[73] Juhani Karhumäki and Svetlana Puzynina. On k-abelian palindromic rich and poor words. In *International Conference on Developments in Language Theory*, pages 191–202. Springer, 2014.

[74] Štěpán Holub and Kalle Saari. On highly palindromic words. *Discrete applied mathematics*, 157(5):953–959, 2009.

[75] Sean Eron Anderson. Bit twiddling hacks. https://graphics.stanford.edu/~seander/bithacks.html, 2005.

[76] Nelson HF Beebe. *The Mathematical-Function Computation Handbook*. Springer, 2017.

[77] Chris Monroe, DM Meekhof, BE King, Wayne M Itano, and David J Wineland. Demonstration of a fundamental quantum logic gate. *Physical review letters*, 75 (25):4714, 1995.

[78] Christof Wetterich. Quantum computing with classical bits. *Nuclear Physics B*, 948:114776, 2019.

[79] Ali Alatabbi, Costas S Iliopoulos, and Mohammad Sohel Rahman. Maximal palindromic factorization. In *Stringology*, pages 70–77, 2013.

[80] Pavlos Antoniou, Maxime Crochemore, Costas S Iliopoulos, Inuka Jayasekera, and Gad M Landau. Conservative string covering of indeterminate strings. In *Stringology*, pages 108–115, 2008.

[81] Stefan Kurtz, Enno Ohlebusch, Chris Schleiermacher, Jens Stoye, and Robert Giegerich. Computation and visualization of degenerate repeats in complete genomes. In *ISMB*, pages 228–238. Citeseer, 2000.

[82] Kevin B Murray, William R Taylor, and Janet M Thornton. Toward the detection and validation of repeats in protein structure. *Proteins: Structure, Function, and Bioinformatics*, 57(2):365–380, 2004.

[83] Sumaiya Nazeen, M Sohel Rahman, and Rezwana Reaz. Indeterminate string inference algorithms. *Journal of Discrete Algorithms*, 10:23–34, 2012.

[84] Dipankar Ranjan Baisya, Mir Md Faysal, Mohammad Sohel Rahman, et al. Degenerate string reconstruction from cover arrays. In *Stringology*, pages 191–205, 2013.

[85] Costas S. Iliopoulos, Ritu Kundu, and Solon P. Pissis. Efficient pattern matching in elastic-degenerate texts. In *Language and Automata Theory and Applications*, pages 131–142. Springer International Publishing, 2017.

[86] Yannis Almirantis, Panagiotis Charalampopoulos, Jia Gao, Costas S Iliopoulos, Manal Mohamed, Solon P Pissis, and Dimitris Polychronopoulos. On avoided words, absent words, and their application to biological sequence analysis. *Algorithms for Molecular Biology*, 12(1):5, 2017.

[87] Roman Kolpakov and Gregory Kucherov. Searching for gapped palindromes. *Theoretical Computer Science*, 410(51):5365–5373, 2009.

[88] Anna E Frid, Svetlana Puzynina, and Luca Q Zamboni. On palindromic factorization of words. *Advances in Applied Mathematics*, 50(5):737–748, 2013.

[89] M Sohel Rahman, Costas S Iliopoulos, and Laurent Mouchard. Pattern matching in degenerate dna/rna sequences. In *WALCOM*, pages 109–120, 2007.

[90] IUPAC encoding. http://www.bioinformatics.org/sms2/iupac.html, 2018.

[91] Randal Cox and Sergei M Mirkin. Characteristic enrichment of dna repeats in different genomes. *Proceedings of the National Academy of Sciences*, 94(10):5237–5242, 1997.

[92] Tonya Woods, Thanawadee Preeprem, Kichun Lee, Woojin Chang, and Brani Vidakovic. Characterizing exons and introns by regularity of nucleotide strings. *Biology direct*, 11(1):6, 2016.

[93] Oluwole Ajala, Miznah Alshammary, Mai Alzamel, Jia Gao, Costas Iliopoulos, Jakub Radoszewski, Wojciech Rytter, and Bruce Watson. On the cyclic regularities of strings. In *IFIP International Conference on Artificial Intelligence Applications and Innovations*, pages 219–224. Springer, 2019.

[94] Ming-Ying Leung, Kwok Pui Choi, Aihua Xia, and Louis HY Chen. Nonrandom clusters of palindromes in herpesvirus genomes. *Journal of Computational Biology*, 12(3):331–354, 2005.

[95] Mikhail S Gelfand and Eugene V Koonin. Avoidance of palindromic words in bacterial and archaeal genomes: a close connection with restriction enzymes. *Nucleic acids research*, 25(12):2430–2439, 1997.

[96] HF Dammers and DJ Polton. Use of the iupac notation in computer processing of information on chemical structures. *Journal of Chemical Documentation*, 8(3): 150–160, 1968.

[97] Andrew D Johnson. An extended iupac nomenclature code for polymorphic nucleic acids. *Bioinformatics*, 26(10):1386–1389, 2010.

[98] Donald J Polton. A computer process for substructure searches on compound structures ciphered in the iupac notation. *Information Storage and Retrieval*, 8 (4):191–201, 1972.

[99] Václav Brázda, Jan Kolomazník, Jiří Lýsek, Lucia Hároníková, Jan Coufal, and Jiří Št'astnỳ. Palindrome analyser–a new web-based server for predicting and evaluating inverted repeats in nucleotide sequences. *Biochemical and biophysical research communications*, 478(4):1739–1745, 2016.

[100] Muthusamy Ramakrishnan, Mingbing Zhou, Chunfang Pan, Heikki Hänninen, Kim Yrjälä, Kunnummal Kurungara Vinod, and Dingqin Tang. Affinities of terminal inverted repeats to dna binding domain of transposase affect the transposition activity of bamboo ppmar2 mariner-like element. *International journal of molecular sciences*, 20(15):3692, 2019.

[101] J John Bissler. Dna inverted repeats and human disease. *Front Biosci*, 3(4): d408–d418, 1998.

[102] Congting Ye, Guoli Ji, Lei Li, and Chun Liang. detectir: A novel program for detecting perfect and imperfect inverted repeats using complex numbers and vector calculation. *PloS one*, 9(11), 2014.

[103] Jialu Hu, Yan Zheng, and Xuequn Shang. Mitefinderii: a novel tool to identify miniature inverted-repeat transposable elements hidden in eukaryotic genomes. *BMC medical genomics*, 11(5):101, 2018.

[104] Yong Wang and Jiao-Mei Huang. Lirex: A package for identification of long inverted repeats in genomes. *Genomics, proteomics & bioinformatics*, 15(2):141–146, 2017.

[105] Reverse-complement tool. http://reverse-complement.com/, n.d. Accessed: 2020-06-16.

[106] Donald J Polton. Conversion of the iupac notation into a form for computer processing. *Information Storage and Retrieval*, 5(1):7–25, 1969.

[107] Zvi Galil and Raffaele Giancarlo. Improved string matching with k mismatches. *ACM SIGACT News*, 17(4):52–54, 1986.

[108] Gad M Landau and Uzi Vishkin. Efficient string matching with k mismatches. *Theoretical Computer Science*, 43:239–249, 1986.

[109] Catherine M Rice, Rainer Fuchs, Desmond G Higgins, Peter J Stoehr, and Graham N Cameron. The embl data library. *Nucleic acids research*, 21(13):2967, 1993.

[110] Dennis A Benson, Ilene Karsch-Mizrachi, David J Lipman, James Ostell, and Eric W Sayers. Genbank. *Nucleic acids research*, 39(Database issue):D32, 2011.

[111] Michelangelo Bucci, Alessandro De Luca, and Gabriele Fici. Enumeration and structure of trapezoidal words. *Theoretical Computer Science*, 468:12–22, 2013.

[112] Golnaz Badkobeh, Hideo Bannai, Keisuke Goto, I Tomohiro, Costas S Iliopoulos, Shunsuke Inenaga, Simon J Puglisi, and Shiho Sugimoto. Closed factorization. *Discrete Applied Mathematics*, 212:23–29, 2016.

[113] GOLNAZ Badkobeh, Gabriele Fici, and Zsuzsanna Lipták. A note on words with the smallest number of closed factors. *CoRR abs/1305.6395*, 2013.

[114] Carl Barton, Costas S Iliopoulos, Solon P Pissis, and William F Smyth. Fast and simple computations using prefix tables under hamming and edit distance. In *International Workshop on Combinatorial Algorithms*, pages 49–61. Springer, 2014.

[115] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *International Symposium on Experimental Algorithms*, pages 326–337. Springer, 2014.

[116] Python. https://www.python.org/, n.d. Accessed: 2020-06-16.

[117] GR Consortium et al. Grch37 genome reference consortium human reference 37 (gca 000001405.1), 2009.

[118] B Murovec, B Stres, et al. Efficient coding of dna. *Acta agriculturae Slovenica (Slovenia)*, 2008.

[119] Michael A Bender and Martin Farach-Colton. The lca problem revisited. In *Latin American Symposium on Theoretical Informatics*, pages 88–94. Springer, 2000.

[120] Mai Alzamel, Panagiotis Charalampopoulos, Costas S Iliopoulos, and Solon P Pissis. How to answer a small batch of rmqs or lca queries in practice. In *International Workshop on Combinatorial Algorithms*, pages 343–355. Springer, 2017.

[121] Maxime Crochemore, Lucian Ilie, Costas S Iliopoulos, Marcin Kubica, Wojciech Rytter, and Tomasz Waleń. Computing the longest previous factor. *European Journal of Combinatorics*, 34(1):15–26, 2013.

[122] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.

[123] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE transactions on Information Theory*, 24(5):530–536, 1978.

[124] SR Kodituwakku and US Amarasinghe. Comparison of lossless data compression algorithms for text data. *Indian journal of computer science and engineering*, 1 (4):416–425, 2010.

[125] Shrusti Porwal, Yashi Chaudhary, Jitendra Joshi, and Manish Jain. Data compression methodologies for lossless data and comparison between algorithms. *International Journal of Engineering Science and Innovative Technology (IJESIT) Volume*, 2:142–147, 2013.

[126] SR Kodituwakku and US Amarasinghe. Comparison of lossless data compression algorithms for text data. *Indian journal of computer science and engineering*, 1 (4):416–425, 2010.

[127] Maxime Crochemore and Lucian Ilie. Computing longest previous factor in linear time and applications. *Information Processing Letters*, 2008.

[128] Supaporn Chairungsee, Tida Butrak, Surangkanang Chareonrak, and Thana Charuphanthuset. Longest previous non-overlapping factors computation. In *2015 26th International Workshop on Database and Expert Systems Applications (DEXA)*, pages 5–8. IEEE, 2015.

[129] Jing He, Hongyu Liang, and Guang Yang. Reversing longest previous factor tables is hard. In *Workshop on Algorithms and Data Structures*, pages 488–499. Springer, 2011.

[130] Hayam Alamro, Lorraine AK Ayad, Panagiotis Charalampopoulos, Costas S Iliopoulos, and Solon P Pissis. Longest common prefixes with k-mismatches and applications. In *International Conference on Current Trends in Theory and Practice of Informatics*, pages 636–649. Springer, 2018.